Abstract of “Static Analyses of Cryptographic Protocols” by Jay McCarthy, Ph.D., Brown University, May 2009

Most protocol analyses only address security properties. However, other properties are important and can increase our understanding of protocols, as well as aid in the deployment and compilation of implementations. We investigate such analyses.

Unfortunately, existing high-level protocol implementation languages do not accept programs that match the style used by the protocol design community. These languages are designed to implement protocol roles independently, not whole protocols. Therefore, a different program must be written for each role. We define a language, WPPL, that avoids this problem. It avoids the need to create a new tool-chain, however, by compiling protocol descriptions into an existing, standard role-based protocol implementation language.

Next, we investigate two families of analyses. The first reveals the implicit design decisions of the protocol designer *and* enables fault-tolerance in implementations. The second characterizes the infinite space of all messages a protocol role *could* accept and enables scalability by determining the session state necessary to support concurrency.

Our entire work is formalized in a mechanical proof checker, the Coq proof assistant, to ensure its theoretical reliability. Our implementations are automatically extracted from the formal Coq theory, so they are guaranteed to implement the theory.

Static Analyses of Cryptographic Protocols

by
Jay McCarthy
B. S., University of Massachusetts at Lowell, 2005
Sc. M., Brown University, 2007

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2009

This dissertation by Jay McCarthy is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date _____ _____

Shriram Krishnamurthi, Advisor

Recommended to the Graduate Council

Date _____ _____

Joshua D. Guttman, Reader
(MITRE Corporation)

Date _____ _____

John Jannotti, Reader
(Brown University)

Date _____ _____

Anna Lysyanskaya, Reader
(Brown University)

Date _____ _____

John D. Ramsdell, Reader
(MITRE Corporation)

Approved by the Graduate Council

Date _____ _____

Sheila Bonde
Dean of the Graduate School

Biography

Jay McCarthy was born in the sleepy town of Dunstable, Massachusetts on the twenty-third of June, 1985. He is the son of a CPA and a software engineering team manager. He has an older brother and sister and a younger sister. He attended the University of Massachusetts in Lowell from September 2001 to May 2005, earning a Bachelor of Science degree in Mathematics and Computer Science, with a Minor in Economics. While at UMass Lowell, he received the Mathematics Achievement Award. The day after finals in 2005, he drove to Providence to start working with Shriram Krishnamurthi in the Computer Science department at Brown. He completed his Masters in May 2007 while studying towards a doctorate in Computer Science at Brown University. He received a National Science Foundation Graduate Research Fellowship in 2006. He is now an assistant professor at Brigham Young University in Provo, Utah.

Acknowledgments

Shriram Krishnamurthi introduced me to real computer science research and taught me valuable lessons about life as an academic. His pearls of wisdom are invaluable, even though he is forced to occasionally cast them before swine.

Joshua Guttman and John Ramsdell have been wonderful in their support and guidance in this project; I would not have done this research if not for them.

Anna and JJ have provided the necessary skepticism on my committee. "Why should I care?" is such an important question, it is vital to have someone who is committed to getting an answer.

I thank the PLT community for their support, advice, and intellectual atmosphere. Special thanks go to Greg Cooper, Matthew Flatt, Matthias Felleisen, Dave Herman, and Guillaume Marceau.

I thank the Brown Ballroom Dance Team for keeping me sane and active my first year at Brown and providing the means by which I met my wife, Elizabeth Day McCarthy.

I thank her for believing in and supporting me through the process: having patience when I needed to focus on writing, reading my inscrutable papers, and waiting for me to move out to live with her.

I thank John "Jack" Day McCarthy for coming to my thesis defense, even though he wasn't born yet.

I am grateful for the financial support of the Computer Science department and the National Science Foundation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

> **Thesis statement.**
> Static analyses of cryptographic protocols can provide insight into the non-security properties of a protocol and aid in the deployment and compilation of implementations. Protocol designers use a specification style that is incompatible with most existing protocol programming languages; a domain-specific language that matches this style will improve the usability of protocol analyses.

We will discuss basic questions that this thesis raises as a way to provide the necessary background and context for this work.

What is a cryptographic protocol?

A *computer protocol* is a syntax and an agreed upon procedure for exchanging information in that syntax.

It is useful to understand protocols by consider what information is exchanged and what parties learn from that exchange. This amounts to considering a *protocol* as a procedure through which parties agree on values and properties of those values. A protocol can be faulty because it fails to deliver agreement. (For example, it possible for a bank to believe that a customer is authorizing a withdrawal when it is actually a thief.)

A *cryptographic protocol* is one that attempts to use cryptographic methods to allow agreement in the presence of an adversary.

We are broad in our notion of protocol. In particular, protocols that fail to *actually* provide agreement in some circumstances are still considered protocols. This is contrary to a substantial literature on "cryptographic protocols" defined as protocols that resist known attacks.

What protocol properties are interesting?

The fundamental part of the thesis is that *non-security* properties of a protocol are worth knowing. Clearly, there are are innumerable such properties, but not all of them are interesting. We have in mind two general

types of properties that are interesting: those that characterize a protocol's operation and construction; and those that correspond to deployment properties.

For example, many protocols can be understood as conglomerations of multiple protocols that perform sub-tasks and interact in a subtle way. However, few protocols are described this way. An analysis that detects this type of structuring would provide insight into the construction of a protocol.

As an example of the second type, not all protocols can be used in unison, because messages of protocol *A* are valid messages of protocol *B*. An analysis that captured the entire space of possible protocol messages would both characterize the protocol *and* enable a further analysis that would correspond to the deployment property of dispatchability, i.e., the lack of confusion between protocol messages.

Analyses of the first type need little justification beyond the principle that understanding protocols further is always good. However, analyses of the second type beg the question, "Why is deployment important?"

Why deployment?

There are many services that stand to benefit from online deployment that require the properties of cryptographic protocols. For example, online banking is an essential feature of any modern bank and there are many banks that exist solely online. Other services, like vehicle registration and similar government procedures, are more convenient when available online. In the future, nearly every service will have an online provider, from baby-sitter location to funeral arrangement.

What is hard about deployment?

Customers are not satisfied with simply "an" online offering. In addition to the inherent benefits of remote connectivity (e.g., convenience and scheduling), the online service must be fast and reliable for every customer. This means the implementation must be scalable. Furthermore, customers know that there are no security cameras and armed guards watching their electronic transactions, so they require some comparable form of security.

These two goals of scalability and security are not easy to satisfy. Each goal has engendered a cadre of techniques and strategies for their achievement. Experience suggests that there are many ways of achieving scalability [101, 141, 111, 137] that are agnostic to the particular ends of the service. However, history is replete with examples of security failures [102, 83, 23, 84, 120] that are intimately tied to the minute details of the protocol. For this reason, there is a large community of researchers developing cryptographic protocols [40, 90, 70, 139] and methods of verifying their security properties [3, 60, 8, 131].

While these two essential components of deployment are hard, there are many good solutions to their subtleties. However, this work has had little broad impact or synthesis. Most deployments are built on only a few security protocols: SSL [40], SSH [146], and Kerberos [75], and a few others. This situation inspires two questions: What is wrong with using just these protocols? And, why aren't the products of security protocol research applied?

The most common protocol cited above, SSL, is inherently a two-party protocol. It establishes a secure session between a client and a server. It provides a means of authenticating each party to the other—although the client authentication is rarely enabled. However, many simple services are inherently about three parties. SSL cannot guarantee a secure session between three parties. Instead, an ad-hoc protocol (perhaps built atop multiple SSL sessions) must be built. This protocol is unlikely to be verified with the same scrutiny as a specialized three-party protocol, or even SSL itself. End-to-end properties and multi-party sessions are at the core of most reasons why the standard protocols are not sufficient for every situation.

We posit a few reasons why verified, specialized protocols are unused in the field. The first reason is that it is difficult for practitioners to know which protocols satisfy their goals, or even what their security goals are. The second reason is that implementations of verified, specialized protocols are rarely available. For example, even a "scalable" protocol from CRYPTO 2003 [70] was not implemented as of January 2008. Even when implementations are available, they are unlikely to meet the scalability expectations of researchers.

This means that practitioners are required to (a) resolve the ambiguities common in informal protocol specifications [9, 8, 12, 2, 11] and (b) risk wasting effort implementing a protocol that will have poor performance characteristics. On the other hand, cryptographic protocol designers are experts in *security*, not *deployment*: therefore, it is unreasonable to rely on their aptitude for protocol implementation.

Where do we put our effort?

In the deployment aspect of our work, we focus on the problem of lack of viable implementations by enabling the production of scalable implementations. The problem that practitioners don't know what protocols are valuable is very real and important. But, it is difficult to address with technical means. It is perhaps best addressed by education and the services of lucratively paid security consultants.

Automated tools can alleviate the specialization mismatch between practitioners and scalable implementations. However, we must deeply understand what scalability goals are important and what special constraints cryptographic protocols impose on any implementations. We cannot simply use off-the-shelf network protocol compilers [74, 95, 76, 5, 125, 59] and hope for the best.

One approach to this problem would be to formally define a "deployment" and deduce what "scalability" and "security" mean. We would effectively be building a model that could accommodate the simulationist definitions of scalability invented by the systems community *and* the definitions of security invented by the protocol design and analysis community. We would then have a model capable of comparing two different deployments, and, therefore, the machinery to automate improvements and optimizations of deployment. This model would also accommodate a definition of an "adversary" of a deployment; we would be capable of determining the capabilities of an adversary to attack both the security *and* scalability properties of the deployment.

However, this is not the direction we have chosen to follow in our research. Instead of deducing the *ur-deployment* and its associated airy fortifications, we proceed on a more low-brow path. Our approach is more

synthetic than analytic. We will assume that existing communities are accurate when expressing their values and do not need to be questioned. We will assume that existing definitions of security and adversary action are satisfactory. We will assume that existing criteria of scalability truly represent the desires of practitioners. We essentially take the perspective of the implementor: How can we faithfully implement a verified protocol in a way that adds scalability, but does not compromise the extant security?

This is where analyses for properties related to deployment enter in: such properties should alleviate the burden of producing such implementations. But what deployment properties are worth investigating? We must understand *what is hard* about deployment to find places where analyses can provide further insight and advantage.

What are the issues?

There are a few reasons why it is difficult to produce scalable implementations of verified protocols. These reasons can be roughly divided into the categories of usability, scalability, and security.

A deployment does not exist in a vacuum; it is part of a larger software and network context. An implementation is only truly usable if it integrates well into this context. The most usable and useful implementation is tailored perfectly to the latest industry fads; e.g., SOAP, WSDL, etc.. A reasonably usable implementation relies only on widely deployed standards; e.g., TCP/IP. Our prototype takes this approach. Our analyses are general enough that it is feasible to tailor the entire system to the latest fad if necessary.

Scalability means something different to nearly every practitioner. For some it means fast recovery from crashes, and for others it means no crashes at all. For some it means minimal per-session state, yet for others it means no per-session state. For some it means throughput maximization over the wire, while for others it means predictable throughput with a perfect 90° "elbow". We interpret scalability as support for many sessions, predictable per-session state, and fault-tolerance through minimal per-session backups.

The security goals of any given protocol are as diverse as scalability expectations. However, we can deal with them in a uniform manner. If we interpret our "security" goal as preservation of the protocol semantics in the implementation, than whatever definition of security is used in the formal proof about a protocol is maintained in our implementation. This technique, however, complicates our implementation. We must ensure that at every step the implementation is faithful to the semantics. This means the implementation cannot rely on network-level services that are, by definition, not part of the protocol, and therefore, not part of the security proof. In particular, this means that we must not rely on the reliability and session properties of TCP/IP. (Although it should be noted that we could *use* them, provided we were not passing on responsibility.) In effect, we must consider the network as a blackboard and recreate the session dispatching features of TCP/IP.

A final issue relevant to our work is the usability of our tool-chain with regard to encoding protocols. We must ensure that our specification language is accessible and does not impose undue burdens on users. For example, many informal protocol specifications contain idiomatic constructions [9, 8, 12, 2, 11], which a naïve user should not be required to elaborate.

How do we model the problem?

Since an essential aspect of our work is the preservation of protocol semantics, it is important to clarify our protocol modeling framework.

At the lowest level, we use the Calculus of Inductive Constructions (CIC), via the Coq proof assistant, as our proof theory and base logic. A posteriori, CIC is sufficiently expressive for the theories we've targeted; a priori, CIC has been used to solve similar programming language/systems problems in the past and has the additional benefit of having an executable kernel. We discuss the pros and cons of CIC in Chapter 2.

We employ strand spaces with trust annotations (embedded in CIC) as the essential theory of cryptographic protocols.

Strand spaces are a succinct and natural language for discussing protocol *inter-action* (as opposed to protocol action); all inessential attributes of protocols are boiled away until all that is left is knowledge and communication bound by casual dependence. In contrast, many other protocol analysis theories are laden with extraneous details about, for example, how knowledge is created or messages constructed, which would only hinder succinct analysis. Furthermore, the strand spaces model is tailored to analyze the powerful Dolev-Yao adversary [35]. This adversary can create, destroy, destructure, modify, or redirect messages without any constraint beyond the assumption of perfect cryptography and a secret kernel of information.

We use an extension of strand spaces that allows rely-guarantee [53] trust management techniques. Rely-guarantee reasoning is a principled way of specifying and verifying interesting protocol goals beyond merely value agreement. It works by introducing the notion of protocol *soundness*: a protocol is sound if whenever a principal assumes something is true, another principal has previously established it. Formally, this means that each communication step of a strand is annotated with a formula representing the trust that is assumed (i.e., relied upon) or established (i.e., guaranteed) at that point. Soundness is thereby a property of a set of communicating annotated strands. This dimension of protocol analysis and verification—commitment and trust—greatly enhances the utility and expressiveness of our model.

We use the CPPL protocol role calculus to specify individual protocol roles. CPPL adds only a bare minimum of baggage atop strands, yet is reasonable for describing real protocols. The strand semantics of CPPL programs is very transparent, and therefore CPPL possesses many of the benefits of strands. CPPL complicates the adversary model by encoding some internal behavior in the strand. An analysis of this strand that does not take this into account can "see into the brain" of the specified role. However, it is easy to squelch this information during analysis and verification.

All of our work builds atop this structure.

We must also utilize a "model" of the typical protocol found in the wild. This model can then be applied to ensure that (a) our specification language can be used to encode the typical protocol, (b) we can gain insight into the typical protocol, and (c) we can achieve our scalability goal with the typical protocol. We will use protocols from the Security Protocols Open Repository [114] (SPORE) and from Clark and Jacob's seminal survey [24] of the protocol literature, collectively, as our model. We discuss the makeup of this test suite in

Chapter 5.

How do we solve the protocol specification problem?

Our specification language, WPPL, is designed with two goals in mind: (1) match the specification style of the protocol literature and (2) have a clear mapping from WPPL phrases to CPPL phrases.

CPPL is used to specify protocols by giving a CPPL interpretation of each protocol role. Each role may duplicate some of the same information, e.g., what messages are sent and received, what trust formulas are guaranteed, etc. In the protocol literature (exemplified by our test suite), protocols are not specified this way. Instead, protocols are specified from the perspective of an omniscient third-party, rather than a collection of first-person accounts.

We have also defined a transformation on WPPL that removes idiomatic constructions from traditional specifications. This ensures that specifications can be used "off the shelf". The essential insight of this transformation is that the omniscient third-party knows "too much" and an account must be derived for each party that *exactly* accommodates his knowledge.

WPPL is compatible with our protocol model. Its semantics is expressed the same way as CPPL: by encoding a set of strand interpretations for each role. Its semantics uses exactly the same strand encoding techniques, so the adversary model is unchanged. However, the omniscient perspective of WPPL suggests more certainty to the reader of the specification than does CPPL; e.g., it obscures the adversary's ability to generated and redirect messages. The central insight of WPPL compilation into CPPL is that the set of communication channels is implicit in WPPL, but explicit in CPPL, so the semantics must track this set during compilation.

How do we support dispatch without reliance on TCP/IP?

Since a fundamental aspect of maintaining the security of deployed protocols is not relying on network-level services that are not verified, an essential scalability feature is to support many concurrent sessions without utilizing the TCP/IP session dispatching feature.

We phrase this problem as a non-security property between two protocols, A and B, (in the case of session dispatching A and B are different sessions of the same protocol; when they are distinct, the property formalizes port dispatching): Is there a message m that would be accepted by some run of A and some run of B?

If this is the case, then there is something fundamental about the protocols that makes it impossible to always dispatch incoming messages without modifying the protocol. (We could, for example, modify the protocols to disallow certain executions of A when certain executions of B are active.)

If this property does not hold, then there is an *inherent* dispatchability between the two protocols, because there are *no* confusing messages.

This property provides insight in its own right, because it provides a formal notion of protocol confusion: a common attack vector. (However, it is not strictly a security analysis, because the presence or lack of message space overlap does not imply any security conclusion.)

We have a verified decision procedure for this property. The essential idea of this analysis is that incoming message patterns describe a set of acceptable messages. The intersection of these two sets is the set of witnesses for the property. They can be generated by comparing two patterns piecewise and eliding certain inessential information—for example, variables are assumed to match, because it is possible to instantiate them to match.

There are two interesting extensions, however. First, distinguishing values, such as randomly generated nonces, are, by definition, unique to a given protocol. Therefore, they can be assumed to never match the unique values of another protocol execution. Second, when comparing a protocol with itself, it is not necessary to consider an initial message pattern, because this pattern represents session *creation* and is therefore only accepted by a single session (the ur-session) at a time.

How do we measure the efficiency of dispatching?

One problem with our dispatching analysis is that it only determines when it is *possible* to distinguishing messages of two protocol executions. It does not tell us how hard it is, how much session state is required, or what secret keys must be used when distinguishing. But, one of our scalability goals is predictable session state.

It is very simple, actually, to determine this when the question is phrased the right way. Suppose that a message pattern was viewed by a party with a limited amount of information, as compared to the context where the pattern is normally seen. Then, there are certain sections of incoming messages that are effectively "black boxes." These components must be assumed to match any (and every) other message pattern. Suppose, however, that two message patterns are *still* distinct with the presence of these black boxes. The session state, or trust given to a dispatcher, is effectively the limited set of information.

We formalize this notion of limited views as message "redaction". An optimization procedure then determines the smallest set, such that two messages are distinguishable given redaction over that set. The key idea of this part of our work is the definition of redaction and recognition that it represents the desired goal. The actual optimization process is extremely naïve, but effective.

The result of this optimization is effectively a way of answering the question: *why* is there a lack of message space overlap between two protocols? Whatever information is redacted does not contribute to distinguishing the two protocols. Therefore, whatever information remains *is*. Thus, this "deployment" property and optimization provides insight into the design of the protocol, if it is not intended to be confused with another, or into an implicit property of the protocol that can provide the basis of further study.

How do we scalably secure implementations against crashes?

A second aspect of session state that is key to scalability is the amount of information that must be reliably saved to recover from crashes. It is always safe to save all information available to a principal as a backup. However, protocol roles are often written such that there are many sub-goals that generate new information

and make obsolete old information. A conservative backup would retain this useless old information, bloat backup media, and ultimately limit the scalability of the service.

Our modeling languages are simple enough that we can calculate the exact (minimal) set of information necessary to continue at any point in a protocol execution. But, there is an important subtlety: Cryptographic protocols are not just about collecting and sharing data; they also manage trust, or formulas about the data. Witnesses of trust guarantees and records of trust assertions are essential parts of protocol backups. If these are not included, then the semantics of the protocol session after restoration is compromised: the proof environment is different and uncertainty reigns.

This minimal backup *is* a representation of what information is *necessary* at a point in the protocol. It therefore demonstrates what is important, and moreover, what is *not*. If we observe the sections of a protocol when a particular value is necessary, then we have a handle on *why* that value is present, or rather, what it is present for. This analysis process provides insight into the explicit (and implicit!) design decisions behind the protocol. It can show how sub-protocols are used within a protocol and how particular guarantees are established. In general, the analysis over minimal backups provide *insight* into the structure of a protocol.

How do we measure the success of this work?

The descriptions above are sufficient to understand the important ideas of our solutions and contributions. But how well do these particular contributions stand-up against our stated goals?

There are two metrics whereby we can measure the success of WPPL: how well it captures the specification style in the literature and whether the compilation to CPPL preserves semantics.

As noted earlier, we model the protocol design literature through our SPORE-based test suite. There are fifty protocols in the suite. Forty-three of these (86%) can be written in WPPL. None of these were sensible, given WPPL well-formedness rules, because of the use of idiomatic constructions. After the idiom removal process is applied, however, every single one was. In other words, the idiom removal process produces canonical, sensible specifications of every protocol that WPPL can express.

The compilation from WPPL to CPPL preserves the semantics of specifications by virtue of a formal, mechanized proof in Coq. There is no ambiguity at all: the compilation is successful.

The correctness, or soundness, of our decision procedure for determining message overlap, and therefore the capability to perform dispatching is without question, due to our use of the Coq proof assistant. However, it is important to look at how effective this procedure is in practice by looking at its results against the test-suite. This tells us how useful the analysis is likely to be and how common these techniques are in the protocol literature.

When we consider every role of the forty-three protocols encoded in WPPL as a candidate for dispatching alongside every other role, we find that 62.1% can be properly distinguished. If we look at each role with itself, i.e., the essential session dispatch scenario, 62.8% roles support multiple sessions.

Like the basic dispatching analysis, the correctness of our redactive optimizer for minimizing the requisite

trust is established by formal proofs. It is also useful in this instance to look at its results with real protocols. We find that 43% of pairs of protocol roles do not require any trust to dispatch, while 18% require the maximal amount of available information. In the case of protocol role sessions, these numbers are respectively 54% and 37%.

We evaluate the effectiveness of our minimal backup calculator in a similar way to the other analyses. The correctness is a straight-forward deduction from our definition of a backup. (The definition captures the correctness condition and the real work is done in the procedure that calculates the backup.) The effectiveness is established by inspecting the behavior of minimal backups at different points in protocol runs. We find that on average, a protocol role will discard over half of its available information (65%) at some point during the run. This means that a fault-tolerance regime built on our minimal backups will be, on average, 65% smaller than a naïve strategy, at the worst point.

We evaluate the analysis of minimal backups by determining if our hypothesis (that minimal backups decrease over time) is true. This corresponds to the efficiency of minimal backups as a fault-tolerance strategy, because that efficiency means that the backup size has decreased. Therefore, protocols *do* have this behavior and our backup-based analysis *can* provide insight into protocol behavior.

What is the impact of this work?

Our work enables the end-to-end production of scalable implementations of cryptographic protocols and provides diagnostic analyses for protocol designers at multiple stages of protocol design and analysis. Figure 1.1 presents the workflow that our system makes possible.

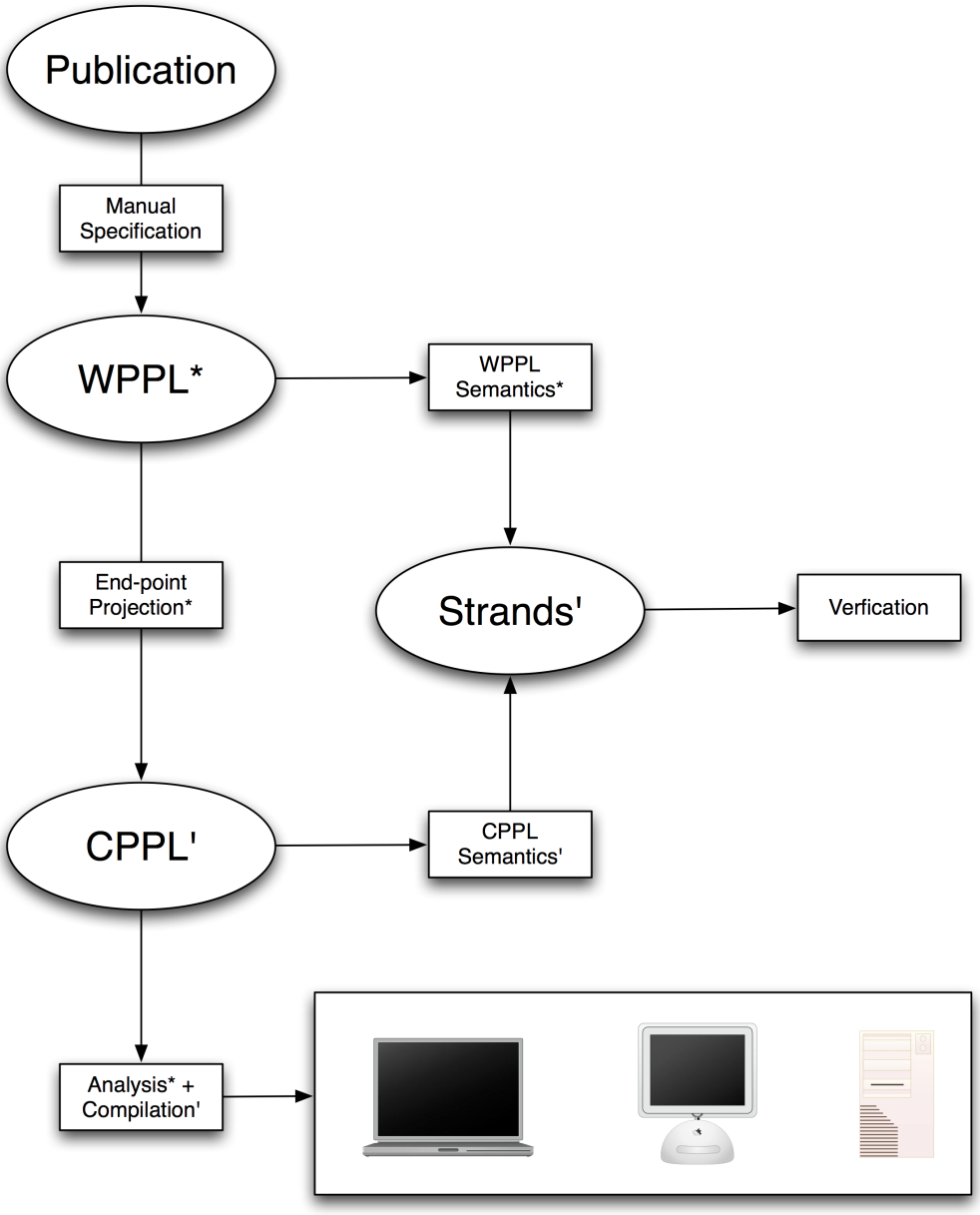
A user can take a protocol specification from, e.g., an existing publication and manually specify it in our domain-specific language, WPPL. This is a relatively pain-free process by the design of WPPL. This specification format is amenable to verification with the standard strand spaces model.

Each role of a WPPL specification can be compiled into a CPPL specification via *end-point projection*. CPPL is an existing domain-specific language for protocol specification that we've modified for our purposes. Its semantics is also given in terms of strand spaces and we formally prove that end-point projection of WPPL specifications preserves meaning.

Once a protocol is specified in CPPL it can be compiled into executable implementations. We've extended the existing compiler to enable greater scalability in implementations—through optimal dispatching and fault tolerance with minimal backups—by incorporate information from new protocol deployment analyses.

Furthermore, when a protocol is specified in CPPL, a protocol designer can compare its message space with other protocols, and is provided with information about why overlap is present (or not). Additionally, a designer (or reverse engineer) can deter the internal structure of the protocol by studying how the minimal backup (necessary information set) changes over time.

This system is an end-to-end solution for compiling protocol specifications into reliable, deployable implementations. In the future, when developers are choosing cryptographic protocols they can use this system



$*$ indicates new work $'$ indicates modification of existing work

Figure 1.1: Workflow

to evaluate (a) the dispatchability constraints of the protocols and (b) the size of backups necessary for fault-tolerance. Furthermore, they can rapidly experiment with many different protocols via the succinct and common specification style of WPPL.

How is this related to the thesis?

Our work affirms the thesis by providing a domain-specific language that matches the style of protocol designers and a suite of analysis that provide insight into non-security properties of a protocol, such as its internal structure and message space. Furthermore, these analyses also enable better deployments and scalable implementations.

Outline

In Part I, We briefly present the background material necessary to understand our technical contributions. we review the Coq proof assistant in Chapter 2, the Strand Spaces protocol model with trust management in Chapter 3, and CPPL in Chapter 4. In Chapter 5, I discuss the test suite of cryptographic protocols used in this work.

In Part II, We move to the first main contribution. In Chapter 6, We describe WPPL, our whole protocol specification language. In this chapter, We describe its semantics, the end-point projection to CPPL, and the idiom removal process that deals with the conventional styles of the protocol design community.

In Part III, We describe our three major analyses. In Chapter 7, We cover the message space characterization problem and provide applications to dispatching and trust optimization. In Chapter 8, the theory of protocol backups is given, along within the procedure to calculate *minimal* backups in our model. We also discover how these backups can be analyzed to gain insight into the structure of a protocol.

Part IV contains the concluding remarks. A general review of related work is covered in Chapter 9, in addition to the individual reviews in each earlier chapter. A speculative plan of future work is discussed in Chapter 10. A final review of our results and contributions is given in Chapter 11.

Part I

Background Material

Chapter 2

The Coq Proof Assistant

It is dangerous to reason loosely and informally about cryptography. In response to this truism, all of the formal material discussed in this dissertation is mechanically formalizing in the Coq proof assistant [134].

Coq provides numerous advantages over paper-and-pencil formalizations. First, Coq can mechanically check our proofs, thereby bestowing much greater confidence on our formalization and on the correctness of our theorems. Second, because all proofs in Coq are constructive, we can automatically extract certified implementations of all our theories. This provides runnable tools (for free!) and give us confidence in the tools as well. Finally, a mechanized representation is more valuable to others who can much more easily adapt our work to related projects and obtain high confidence in the results.

In the body of this dissertation, proofs will generally not be given in-line, either as Coq or prose. Instead, we will informally describe the structure of the proof when it is interesting, but otherwise refer the reader to the Coq proof scripts. In this chapter, I give a brief overview of the meta-theory of Coq.

2.1 What is a proof?

A formal proof is a rigorous deduction of a theorem from axioms and primitive rules. Common proofs that we typically find in textbooks and scientific papers are informal: they are high-level sketches that could, in theory, be turned into formal proofs by experts. (See Millo [100] for a good discussion of the difference.)

There are many formalisms for expressing formal proofs and their correctness. Many are based on typed λ-calculi. According to Curry-Howard isomorphism [63], a λ-calculus term is a proof of the proposition associated with its type. This is because it is a witness that the type is inhabited.

As an example, consider the proposition $\forall\alpha.\alpha \implies \alpha$. One λ term with this type is $\lambda x.x$. A type-checker runs on this λ term to calculate its type, then compares the result with the proposition in question. If they match, then the proposition is true. As a type system becomes more complicated, more propositions can be expressed in the type language and proved in the term language. The main problem with developing type

systems for this purpose is ensuring that the typing relation is decidable.

The logical framework used by Coq is the Calculus of Inductive Constructions: a dependently-typed λ-calculus with well-founded (co)induction. Coq implements a type-checker for this language and provides tools for building proof terms. The only part of Coq that requires trust is the type-checking kernel. All of the tools merely construct terms that must be checked.

For a simple idea of how Coq works, consider this interaction from the online tutorial:

```
Coq < Variables A B C : Prop.
```

We define some variables with a type of "proposition".

```
Coq < Goal (A -> B -> C) -> (A -> B) -> A -> C.

  A : Prop
  B : Prop
  C : Prop
  ============================
   (A -> B -> C) -> (A -> B) -> A -> C
```

We inform Coq that we would like to prove a particular theorem. Coq displays this as a stack of assumptions and a goal.

```
Coq < intros H H' HA.

  A : Prop
  B : Prop
  C : Prop
  H : A -> B -> C
  H' : A -> B
  HA : A
  ============================
   C
```

Whenever the goal is of the form $\alpha \implies \beta$, we can use the deduction theorem and introduces α as a new assumption and move forward with β as the new goal. We can string any number of these introductions together.

```
Coq < apply H.
```

At this point, we see that H implies the goal. So we "apply" H.

```
2 subgoals

  A : Prop
  B : Prop
  C : Prop
  H : A -> B -> C
  H' : A -> B
  HA : A
  ============================
   A
subgoal 2 is:
 B
```

This creates two new goals, because *H* has two assumptions that must be proven. The rest of the proof is straight-forward:

```
Coq < exact HA.

  A : Prop
  B : Prop
  C : Prop
  H : A -> B -> C
  H' : A -> B
  HA : A
  ============================
   B

Coq < apply H'.

  A : Prop
  B : Prop
  C : Prop
  H : A -> B -> C
  H' : A -> B
  HA : A
  ============================
   A
```

```
Coq < exact HA.
Proof completed.
```

2.2 Program Extraction

The Coq type system separates types into two kinds (types-of-types): Prop and Set. Prop represents logical propositions, such as "n is even" or "there exists an n, such that n is greater than m". Set represents values, such as "a natural number" or "a natural number n such that $P(n)$ is true". (Notice the difference between "exists" and "such that": one is a proposition, while the other is a value with a proposition.)

Sub-terms with types of kind Prop can be erased to reduce a logical term to a value term, or a program that computes with only values. For example, "a natural number n such that $P(n)$ is true" is reduced to "n" (such that $P(n)$ is true), while "n is even" is reduced to "", i.e., it is erased completely. This is called program extraction. Terms that only use Set types can be directly transcribed into Haskell, ML, or Scheme and run with traditional systems.

Since proofs are terms, they can be extracted as well. A compelling example of this is compilation by proof. Suppose we have languages L_0 and L_1, where $\ell \downarrow_x v$ means that the L_x program ℓ evaluates to v. A proof of the theorem, "For all L_0 programs ℓ_0, where $\ell_0 \downarrow_0 v$, there is an L_1 program ℓ_1, such that $\ell_1 \downarrow_1 v$", when extracted *is* a compiler: a function from programs of L_0 to programs of L_1, such that the meaning is the same.

We use this method of software construction almost exclusively in our work. (The exceptions are parsing, printing, and compilation of CPPL to ML.)

2.3 Complications

No theorem prover is perfect and Coq is no exception. Many users of these systems have war stories about formalizing an existing theory with its proofs and the necessity of re-engineering the proofs from the ground up, because the traditional proof strategies are too complicated to encode. We did not have this particular problem, because we had no existing theory. However, we did have a few problems.

First, dealing with program extraction can be very complicated. It takes a concerted effort to use "such that" rather than "there exists" and oftimes one extractable proof will require earlier theorems to be converted to an extractable form. This can be very tedious and sometimes difficult, particularly when a standard library theorem must be converted. Another aspect of program extraction that is complicated is the disallowance of common proof strategies, such as inversion on Prop hypotheses. Since these are erased, they cannot be analyzed. This can lead to some very backwards looking proofs.

Second, mutual induction is very hard to organize correctly. The induction principle generated by Coq isn't quite right and it is often simpler to just embedded one definition within another than deal with correcting

Coq. In our particular circumstances, the historical definition of CPPL contains a list of continuations within the continuation datatype. Since both a list and a continuation is an inductive datatype, there is mutual induction. In my definition, I have linearized the continuation datatype and removed the list instance to avoid this problem. (This actually increases the expressiveness of Coq, so it is a win-win.)

2.4 Conclusion

This introduction to Coq has been intentionally brief. Any conscientious student of Coq must consult the online documentation and reference manuals. However, we will not require any detailed knowledge of Coq in the remainder of the dissertation.

Chapter 3

Strand Spaces with Trust Management

Before we can adequately talk about our language, we must define our goals explicitly. First, what is a cryptographic protocol and how can it be considered cryptographically trustworthy or sound? In this chapter, we define cryptographic protocols and their soundness following the strand space model.

The essence of a cryptographic protocol is a list of messages that are sent and received. These messages are decorated with the trust management formulas that they rely on or guarantee. This intuition precisely matches the extant *strand space* [131] model of cryptographic protocols extended with trust managment formulas [53]. We will use this model throughout our work.

3.1 Syntax

Our syntax of strands is presented in Figure 3.1. Figure 3.2 presents the strand encoding of role `B` of the Andrew Secure RPC [122] protocol. This strand only encodes half of the protocol; another strand would be necessary to fully encode the protocol.

A strand (s) is a list of strand descriptions (sd), where "." represents the empty strand and $sd \rightarrow s$ represents combining one description (sd) with the rest of the strand (s). A strand description is either: $+m, \Phi$, which represents guaranteeing the formulas Φ and sending m; or, $-m, \Psi$, which represents receiving the message m and relying on the formulas Ψ.

Formulas in relies and guarantees may contain strand identifiers, in addition to logical variables and strand values. If bound in the environment at runtime, a strand identifier will be replaced in Φ (Ψ) by the value to which it is bound; if not yet bound, it serves as a query variable that will be bound as a consequence of a trust management call. Logical variables in a trust management formula, if they occur, are interpreted implicitly universally.

$$
\begin{array}{rcl}
s & \rightarrow & . \quad | \quad sd \rightarrow s \\
sd & \rightarrow & +m, \Phi \quad | \quad -m, \Psi \\
m & \rightarrow & \text{nil} \quad | \quad v \quad | \quad k \\
 & | & (m, m') \quad | \quad hash(m) \quad | \quad < v = m > \\
 & | & [m]v \quad | \quad [|m|]v \quad | \quad \{m\}v \quad | \quad \{|m|\}v \\
v, u & \rightarrow & x : t \\
t & \rightarrow & \text{text} \quad | \quad \text{msg} \quad | \quad \text{nonce} \\
 & | & \text{name} \quad | \quad \text{symkey} \quad | \quad \text{pubkey} \quad | \quad \text{channel}
\end{array}
$$

Figure 3.1: Strand Spaces Syntax

```
0  - ("call", pr:name, "b", ai:nonce, a:name, b:name, kab:symkey), _ →
1  + ("accept"), accept([chana:channel]) →
2  - ("msg", chana, a, {| na:nonce |} kab), _ →
3  + ("new_nonce"), new_nonce([nb:nonce]) →
4  + ("msg", chana, {| na, nb |} kab), _ →
5  - ("msg", chana, {| nb |} kab), _ →
6  + ("new_nonce"), new_nonce([nbn:nonce]) →
7  + ("new_symkey"), new_symkey([kabn:symkey]) →
8  + ("msg", chana, {| kabn, nbn |} kab), _ →
9  + ("ret", ai, kabn), _ → .
```

Figure 3.2: Andrew Secure RPC Role `B` Strand

We use m for message patterns. They may be constructed by concatenation (,), hashing ($hash(m)$), variable binding and pattern matching ($< v = m >$), asymmetric signing ($[m]v$), symmetric signing ($[|m|]v$), asymmetric encryption ($\{m\}v$), and symmetric encryption ($\{|m|\}v$). In these last four cases, v is said to be in the *key-position*. For example, in the Andrew role `kab` is in key-position on line 2. Concatenation is right associative. Parentheses ("(" and ")") are informally used to control precedence.

3.2 Semantic Interpretation

A strand merely specifies what messages are sent and received. In this model, a protocol designer regards the cryptographic primitives as black boxes, and concentrates on the structural aspects of the protocol. The representation does not specify to whom messages are sent or from whence they are received. This corresponds to the Dolev-Yao model [35] that allows an adversary to have maximal power to manipulate the protocol by modifying, redirecting, and generating messages *ex nihilo*. This ensures that proofs built on the semantics are secure in the face of a powerful adversary.

The basic abilities of adversary behavior that make up the Dolev-Yao model including transmitting a

known value, such as a name, a key, or whole message; transmitting an encrypted message after learning its plain text and key; and transmitting a plain text after learning a ciphertext and its decryption key. The adversary can also manipulate the plain-text structure of messages: concatenating and separating message components, adding and removing constants, etc. Since an adversary that encrypts or decrypts must learn the key from the network, any key used by the adversary—compromised keys—have always been transmitted by some participant.

A useful concept when discussing the adversary is a *uniquely originating value*. This is a value that only originates (enters the network) at one unique location. Nonces and other randomly generated data are perfect examples of unique origination. By definition, the adversary cannot know these values until they have sent in an unprotected context.

A strand is *local* in the sense that it describes what one principal P does. This involves deciding what values to bind to identifiers; what messages to send; how to process a message that is received; and how to select a procedure to call as a sub-protocol. A strand says nothing about how messages are routed on a network; nothing about what another principal P' does with messages received from P; nothing about how another principal P'' created the messages that P receives; etc. Likewise, it describes only the execution of one protocol role. In essence, the execution semantics describes only a single principal executing a single run of a single role.

To reason about a protocol's execution as a whole, we need to combine strands. We do this using the *global* bundle semantics as provided by earlier work [48]. But, the details of this are not important for our purposes, because our work naturally focuses on the local semantics, i.e., the execution at a single server, rather than the global semantics. However, it is incredibly important when actually using strand spaces to verify protocols to include this step in the analysis.

3.3 The Runtime Environment

To explain how strands execute, we must first introduce the notion of a runtime environment. In our view of protocol behavior, as a principal executes a single local run of a protocol, it builds up an *environment* that binds identifiers to values encountered. As in logic programming, these bindings are commitments, never to be updated; once a value has been bound to an identifier, future occurrences of that identifier must match the value or else execution of the run aborts. For example, on line 2 of the Andrew role, the value for `a` must be the same as was received on line 0; any other value will prevent execution of the run from continuing.

We also have an auxiliary notion of *guaranteeing* formulas Φ in a runtime environment. This means asking the runtime trust management system to attempt to prove the formulas Φ. This occurs on line 1 of the Andrew role (Fig. 3.2), where the formula `accept([chana:channel])` is guaranteed. Identifiers in Φ already bound in the runtime environment are instantiated to the associated values. Identifiers not yet bound in the runtime environment are instantiated by the trust management system, if possible, to values that make

the formulas Φ true. Thus, on line 1 of the Andrew role, a new channel is instantiated to accept a connection. The runtime environment extended with these new bindings is the result of successfully guaranteeing Φ. Thus, on line 1 of the Andrew role, the identifier `chana` is bound. If the runtime trust management system fails to establish an instance of Φ the guarantee fails. This notion of guaranteeing is employed whenever a message is sent.

Finally, there is the notion of *relying* on formulas Ψ in a runtime environment. This means adding assumptions (or "facts" in Datalog terminology) to the runtime trust management system. As with guarantees, identifiers in Ψ must have the same values as those in the runtime environment. These facts are available for future instances of guaranteeing. The Andrew role (Fig. 3.2) does not contain any relies. However, if on line 2 there were a formula, it would be relied upon. In general, relying is employed whenever a message is received.

These trust management notions coalesce into the crucial definition of protocol *soundness*: a protocol is sound if whenever a principal relies on a formula, another principal has previously guaranteed it. This dimension of protocol analysis and verification—commitment and trust—greatly enhances the utility and expressiveness of strands. We, however, do not delve deep into these details, as security analysis is outside the scope of our work. (In particular, most protocols we study were *not* designed using trust management formulas to expressed their goals.)

3.4 Well-formedness

Compilers use BNF context-free grammars to syntactically parse programs. They must also use context-sensitive rules to determine which syntactic expressions are legal programs: e.g., when the syntax refers only to bound identifiers. Similarly, not all strands describe realizable protocols. For example, $(+x, \_ \rightarrow .)$ cannot be realized, because it does not account for where the value for x comes from. Like a programming language, every identifier in a strand must be bound for it to be well-formed.

The only surprising aspect of the well-formedness condition on strands is that, due to cryptographic primitives, the conditions are different on message patterns used for sending and receiving.

Intuitively, to send a message we must be able to construct it, and to construct it, every identifier must be bound. Therefore, a pattern m is well-formed for sending in an environment σ (referred to as `type_msg_send` σ `m` in the Coq specification, but written $\sigma \vdash_s m$ herein; see Figure 3.3) if all identifiers that appear in it are bound. For example, the message on line 4 of the Andrew role is well-formed, but if we removed line 3 it would not be, because `nb` would not be bound.

A similar intuition holds for using a message pattern to receive messages. To check whether a message matches a pattern, the identifiers that confirm its shape—namely, those that are used as keys or under a **hash**—must be known to the principal. Thus, we define that a pattern m is well-formed for receiving in an

$$\textsc{Nil}\ \frac{}{\sigma \vdash_s nil} \qquad \textsc{Var}\ \frac{v \in \sigma}{\sigma \vdash_s v} \qquad \textsc{Const}\ \frac{}{\sigma \vdash_s k} \qquad \textsc{Join}\ \frac{\sigma \vdash_s m \qquad \sigma \vdash_s m'}{\sigma \vdash_s (m, m')} \qquad \textsc{Hash}\ \frac{\sigma \vdash_s m}{\sigma \vdash_s hash(m)} \qquad \textsc{Bind}\ \frac{\sigma \vdash_s m}{\sigma \vdash_s < (i : msg) = m >}$$

$$\textsc{SymEnc}\ \frac{i \in \sigma \qquad \sigma \vdash_s m}{\sigma \vdash_s \{|m|\}(i : symkey)} \qquad \textsc{SymSign}\ \frac{i \in \sigma \qquad \sigma \vdash_s m}{\sigma \vdash_s [|m|](i : symkey)} \qquad \textsc{PubEnc}\ \frac{i \in \sigma \qquad \sigma \vdash_s m}{\sigma \vdash_s \{m\}(i : pubkey)} \qquad \textsc{PubSign}\ \frac{i \in \sigma \qquad \sigma \vdash_s m}{\sigma \vdash_s [m](i : pubkey)}$$

Figure 3.3: Message well-formedness (sending)

$$\textsc{Nil}\ \frac{}{\sigma \vdash_r nil} \qquad \textsc{Var}\ \frac{}{\sigma \vdash_r v} \qquad \textsc{Const}\ \frac{}{\sigma \vdash_r k} \qquad \textsc{Join}\ \frac{\sigma \vdash_r m \qquad \sigma \vdash_r m'}{\sigma \vdash_r (m, m')} \qquad \textsc{Hash}\ \frac{\sigma \vdash_s m}{\sigma \vdash_r hash(m)} \qquad \textsc{Bind}\ \frac{\sigma \vdash_r m}{\sigma \vdash_r < (i : msg) = m >}$$

$$\textsc{SymEnc}\ \frac{i \in \sigma \qquad \sigma \vdash_r m}{\sigma \vdash_r \{|m|\}(i : symkey)} \qquad \textsc{SymSign}\ \frac{i \in \sigma \qquad \sigma \vdash_r m}{\sigma \vdash_r [|m|](i : symkey)} \qquad \textsc{PubEnc}\ \frac{i \in \sigma \qquad \sigma \vdash_r m}{\sigma \vdash_r \{m\}(i : pubkey)} \qquad \textsc{PubSign}\ \frac{i \in \sigma \qquad \sigma \vdash_r m}{\sigma \vdash_r [m](i : pubkey)}$$

Figure 3.4: Message well-formedness (receiving)

environment σ (written `type_msg_recv` σ `m` and $\sigma \vdash_r m$; see Figure 3.4) if all identifiers that appear in key-positions or **hash**es are bound. For example, the message pattern on line 2 of the Andrew role is well-formed because `kab` is bound, but if it were not, then the pattern would not be well-formed.

We write $\sigma \vdash fs$ to mean that the formulas fs are well-formed in the environment σ. This holds exactly when the identifiers mentioned in fs are a subset of σ.

We write $\sigma \vdash sd$ to mean that the strand description sd is well-formed in the environment σ.

$\sigma \vdash +m, \Phi$ holds if and only if $\sigma \vdash \Phi$ and $\sigma \cup bound(\Phi) \vdash_s m$; this corresponds to the intuition that Φ is guaranteed and *then* the message m is sent.

$\sigma \vdash -m, \Psi$ holds if and only if $\sigma \vdash_r m$ and $\sigma \cup bound(m) \vdash \Psi$; this corresponds to the intuition that m is received and *then* Ψ is relied upon.

We write $\sigma \vdash s$ to mean that the strand s is well-formed in the environment σ.

$\sigma \vdash .$ holds for all σ.

$\sigma \vdash sd \rightarrow s$ holds if and only if $\sigma \vdash sd$ and $\sigma \cup bound(sd) \vdash s$, corresponding to the intuition that strands describe an order of message reception and transmission.

3.5 Related Work

The Strand Spaces model is not our original work. We choose it our as verification framework due to its connection with CPPL (discussed in Chapter 4). This model has a rich body of analysis research [132, 133, 49, 45, 44, 50, 46, 53, 52, 34], as well as formal relations to many other verification methods [128, 22, 55, 29, 18, 57, 21], which our work can seamlessly leverage for verification purposes. Therefore, our choice of the strand spaces model is not short-sighted.

While on its own, strands do not solve any of the particular problems of our work—deploying cryptographic protocols—they perform an essential task in the "big picture." Any security protocol would be useless if it didn't actually provide security. Since our system is based on strands, it is natural to ascertain whether or not any given protocol actually meets its goals. Therefore, strands are essential.

3.6 Conclusion

Strand spaces are a convenient and simple notation for specifying cryptographic protocols. However, no one would actually write down the strand semantics of a protocol initially. In the next chapter, we discuss a domain-specific language that is specifically tailored for specifying protocols with a strand space interpretation.

Chapter 4

The Cryptographic Protocol Programming Language

In the last chapter, we discussed a formal model of cryptographic protocols. Naturally, it is not feasible for practitioners to directly use this model. We use CPPL to address this problem.

CPPL [48] is a domain-specific language for expressing cryptographic protocols with trust annotations. The definition we use slightly extends the original work with stand-alone trust management database interaction, message let values, empty messages, and failure continuations. CPPL allows the programmer to control protocol actions using trust constraints [53], so that an action such as transmitting a message will occur only when the indicated trust constraint is satisfied. The CPPL semantics identifies a set of *strands* [131] annotated with trust formulas and the values assumed to be unique, as the meaning of a role in a protocol.

4.1 Syntax

The syntax of the CPPL core language is presented in Figure 4.1. Figure 4.2 shows the CPPL encoding of the Andrew B role from Ch. 3. The CPPL core language has procedure declarations and eight types of code statements. It uses the same syntactic conventions as strands.

A procedure declaration specifies the name x of the procedure, a list v^* of formal parameters, and a list of preconditions Ψ involving the formal parameters. The body of the procedure is a code statement c. We will interchangeably refer to procedures as "programs." A code statement may be: failure, a return instruction, which specifies a list of postconditions Φ and return parameters (v^*); a let-statement; a trust management interaction; a send, a receive, a call, or a match.

$$
\begin{array}{rcl}
p & \rightarrow & \text{proc } x\ v^*\ \Psi\ c \\
c & \rightarrow & \text{fail} \quad | \quad \text{return } \Phi\ v^* \\
 & | & \text{let } v = lv \text{ in } c \quad | \quad \text{derive } \Phi\ c\ c' \\
 & | & \text{send } \Phi\ v\ m\ c\ c' \quad | \quad \text{recv } v\ m\ \Psi\ c\ c' \\
 & | & \text{call } \Phi\ x\ v^*\ u^*\ \Psi\ c\ c' \quad | \quad \text{match } v\ m\ \Psi\ c\ c' \\
lv & \rightarrow & \text{new nonce} \quad | \quad \text{new symkey} \quad | \quad \text{new pubkey} \\
 & | & \text{accept} \quad | \quad \text{connect } v \quad | \quad m
\end{array}
$$

Figure 4.1: CPPL Syntax

```
 1 proc b (a:name b:name kab:symkey) _
 2  let chana = accept in
 3  recv chana (a, {| na:nonce |} kab) -> _ then
 4   let nb = new nonce in
 5   send _ -> chana {| na, nb |} kab then
 6   recv chana {| nb |} kab -> _ then
 7    let nbn = new nonce in
 8    let kabn = new symkey in
 9    send _ -> chana {| kabn, nbn |} kab then
10    return _ (kabn)
11    else fail
12   else fail else fail
13  else fail
14 end
```

Figure 4.2: Andrew Secure RPC Role B in CPPL

4.2 Informal Execution Semantics

The execution of a **fail** statement is an unconditional failure. The execution of a **return** statement attempts to guarantee the formulas Φ, and then returns the parameters v^* from the resulting extended environment. The execution of a **let** statement depends on the right-hand side: either a new value is generated of the appropriate type; or, a channel is accepted; or, a connection is made to a particular name; or, a message is bound to the variable on the left-hand side. The execution of an **derive** statement attempts to guarantee the formulas Φ; if successful, c is executed, otherwise, c' is. The execution of a **send** statement attempts to guarantee the formulas Φ; if successful, message m is sent to the channel v and c is executed, otherwise, c' is. The execution of a **recv** statement attempts to receive the message m from the channel v; if successful, the formulas Ψ are relied upon and c is executed, otherwise, c' is. The execution of a **call** statement is treated as a **send** followed by a **recv**. The execution of a **match** statement attempts to match the message m against v; if successful, the formulas Ψ are relied upon and c is executed, otherwise, c' is.

$$\textsc{Nonce}\ \frac{}{\sigma \vdash \text{let } (v\text{:nonce}) = \text{new nonce}} \qquad \textsc{Symkey}\ \frac{}{\sigma \vdash \text{let } (v\text{:symkey}) = \text{new symkey}} \qquad \textsc{Pubkey}\ \frac{}{\sigma \vdash \text{let } (v\text{:pubkey}) = \text{new pubkey}}$$

$$\textsc{Accept}\ \frac{}{\sigma \vdash \text{let } (v\text{:channel}) = \text{accept}} \qquad \textsc{Connect}\ \frac{nv \in \sigma}{\sigma \vdash \text{let } (v\text{:channel}) = (nv\text{:name})} \qquad \textsc{Msg}\ \frac{\sigma \vdash_s m}{\sigma \vdash \text{let } (v\text{:msg}) = m}$$

Figure 4.3: CPPL let well-formedness

4.3 Well-formedness

CPPL procedures and code statements are well-formed when all identifiers used are bound. This in turn means that messages are well-formed in their appropriate context: sending or receiving. We write $\sigma \vdash c$ to mean that code statement c is well-formed in σ. Similarly, we write $\vdash p$ for well-formedness of procedures. See Figures 4.3, 4.4, and 4.5.

Additionally, CPPL procedures are only well-formed if there is at least one **return** statement and that every **return** statement guarantees the same formulas and returns the same variables. This (essentially) ensures that the procedure has a single domain in all execution paths.

4.4 Semantics

The semantics of a CPPL phrase is given by a set of *strand*, each of which describes one possible local run.

We write $c \to s, \nu$ to codify that the strand s describes the code statement c, under the assumption that the identifiers in ν are uniquely originating (see Sec. 3.2). A single code statement could be described by many different pairs of strands and ν sets. We write $p \to s, \nu$ to mean that a procedure p is described by the strand s with the unique identifiers ν. As with code statements, procedures can have many strand descriptions.

This semantics is completely specified by the inductive definitions in the Coq sources. The semantics for code statements is called `eval_cont` and the semantics for procedures is called `eval_proc`. Both are located in the `CPPL.v` file. We reformat them in Figures 4.6 and 4.7.

The most important theorem about this semantics is that if a procedure is well-formed, then the strand it denotes is well-formed:

Theorem 1. *(`eval_proc_type`) If* $\vdash p$ *then if* $p \to s, \nu$*, then* $\emptyset \vdash s$*.*

Proof summary. We prove this by induction on the code statement component of the procedure p, then by inverting the evaluation derivation. Each case is dispatched by simple set theory solvers and lemmas. □

Adversary. The adversary in the CPPL semantics is not appreciably different for the standard strand spaces/Dolev-Yao adversary: all capabilities and strategies are maintained. However, our encoding of the strand semantics

$$\textsc{Fail}\ \frac{}{\sigma \vdash \text{fail}} \qquad \textsc{Return}\ \frac{\sigma \vdash \Phi \qquad v^* \subseteq \sigma \cup \mathit{bound}(\Phi)}{\sigma \vdash \text{return } \Phi\ v^*} \qquad \textsc{Let}\ \frac{\sigma \vdash \text{let } v = lv \qquad \sigma \cup \{v\}}{\sigma c \vdash \text{let } v = lv \text{ in } c}$$

$$\textsc{Derive}\ \frac{\sigma \vdash \Phi \qquad \sigma \cup \mathit{bound}(\Phi) \vdash c \qquad \sigma \vdash c'}{\sigma \vdash \text{derive } \Phi\ c\ c'}$$

$$\textsc{Send}\ \frac{\sigma \vdash \Phi \qquad v \in \sigma \cup \mathit{bound}(\Phi) \qquad \sigma \cup \mathit{bound}(\Phi) \vdash_s m \qquad \sigma \cup \mathit{bound}(\Phi) \vdash c \qquad \sigma \vdash c'}{\sigma \vdash \text{send } \Phi\ (v\text{:channel})\ m\ c\ c'}$$

$$\textsc{Receive}\ \frac{v \in \sigma \qquad \sigma \vdash_r m \qquad \sigma \cup \mathit{bound}(m) \vdash \Psi \qquad \sigma \cup \mathit{bound}(m) \cup \mathit{bound}(\Psi) \vdash c \qquad \sigma \vdash c'}{\sigma \vdash \text{recv } (v\text{:channel})\ m\ \Psi\ c\ c'}$$

$$\textsc{Call}\ \frac{\sigma \vdash \Phi \qquad v^* \subseteq \sigma \cup \mathit{bound}(\Phi) \qquad \sigma \cup \mathit{bound}(\Phi) \cup u^* \vdash \Psi \qquad \sigma \cup \mathit{bound}(\Phi) \cup u^* \cup \mathit{bound}(\Psi) \vdash c \qquad \sigma \vdash c'}{\sigma \vdash \text{call } \Phi\ x\ v^*\ u^*\ \Psi\ c\ c'}$$

$$\textsc{Match}\ \frac{v \in \sigma \qquad \sigma \vdash_r m \qquad \sigma \cup \mathit{bound}(m) \vdash \Psi \qquad \sigma \cup \mathit{bound}(m) \cup \mathit{bound}(\Psi) \vdash c \qquad \sigma \vdash c'}{\sigma \vdash \text{match } (v\text{:msg})\ m\ \Psi\ c\ c'}$$

Figure 4.4: CPPL code statement well-formedness

of a CPPL phrase includes "internal dialogue". In the above example (binding a new nonce), a message is emitted to account for the introduction of a new value in the strand semantics. This is not a real message that is sent to the outside world. In the case of pattern matching, the message to be matched is emitted, then a message matching the pattern is expected. Again, these are not actual communications. All real "external" messages are tagged in the strands generated by the CPPL semantics, so they can be automatically extracted and analyzed separately. It is conceivable, however, to study the entire strand from a security perspective to understand a protocol's operation on a hostile host. We do not consider this in our work.

4.5 Original Semantics

The semantics in Section 4.4 is different than the published CPPL semantics [48]. Some of these differences are because we have extended CPPL with additional constructs, while others are fundamental. In this section we discuss the fundamental differences between our semantics and the original semantics.

Strand Descriptions. In the original work, an entity called a "strand description" is used to describe the strands that are the interpretations of CPPL phrases. The strand descriptions used therein effectively allowed

$$\textsc{Proc}\ \frac{\sigma \cup v^* \cup \mathit{bound}(\Psi) \vdash c}{\vdash \text{proc } x\ v^*\ \Psi\ c}$$

Figure 4.5: CPPL procedure well-formedness

the authors to attach trust management annotations to the local steps of the strand space for the protocol role.

In our work, we described "strands" (in Ch. 3) as if they *were* these strand descriptions. Since we do not ever "go beneath" these annotated strands, they are sufficient for our purposes, so we abuse terminology and avoid the cognitive energy expended in the additional layer.

This is a mostly trivial difference in the two semantics, but is important to acknowledge when reading the two texts.

Judgments. Our semantic judgment takes the form $c \rightarrow s, \nu$, where c is a CPPL phrase, s is an annotated strand and ν is a set of identifiers whose values will be assumed to be unique.

The original judgment has the form $\sigma, \Gamma \vdash c : s', \nu$, where σ is a runtime environment (a finite function mapping identifiers to values), Γ is a set of formulas serving as a trust management theory, c is a CPPL phrase, s' is a strand description and ν is a set of unique values. The original semantics has auxiliary judgments: $\Gamma \rightarrow \phi$ to mean that the formula ϕ is a logical consequence of the formulas Γ and $\Gamma \Vdash \Phi$ to be this extended to a list of formulas Φ.

Ramifications. Our semantics is essentially a compiler from CPPL into an unexecuted annotated strand notation. In principal, we could describe the execution of these strands by a process where as the role executes and interacts with the outside world, possible strands are eliminated one-by-one, until the actual behavior is revealed. However, we do not do this. Instead, our semantics only provides the set of possible behaviors, or outcomes of an execution.

In contrast, the original semantics represents the execution of the CPPL phrase. The runtime environment and trust management theory are constructed and retained to direct this process. As the execution proceeds, a trace of the execution (described by a strand) is produced for analysis purposes.

Since our semantics is not actually an execution, we do not include the runtime environment in the judgment, nor do we include the trust management theory or its judgments. Any actions of the original semantics that modify these must, however, be accounted for in our semantics. For example, when a fresh value is generated, the runtime environment needs to be expanded. In the original semantics, this is done directly, while in our semantics it is done implicitly by including "internal dialogue": an empty transmission with a trust management query that binds the appropriate identifier.

Since the original semantics is an execution, every strand description contains concrete values, while ours contain identifiers that stand in for values. (However, the original work does define "parametric strands" that

FAIL
$c \to (+\texttt{fail}, ai, \_), \emptyset$

RETURN
$\text{return } \Phi\ x^* \to (+\texttt{return}, ai, x^*, \Phi), \emptyset$

LET (NEW NONCE)
$$\frac{c \to s, \nu}{\text{let } x = \text{new nonce in } c \to (+\texttt{new\_nonce}, \{new\_nonce(?x)\} \to s), \{x\} \cup \nu}$$

LET (NEW SYMKEY)
$$\frac{c \to s, \nu}{\text{let } x = \text{new symkey in } c \to (+\texttt{new\_symkey}, \{new\_symkey(?x)\} \to s), \{x\} \cup \nu}$$

LET (NEW PUBKEY)
$$\frac{c \to s, \nu}{\text{let } x = \text{new pubkey in } c \to (+\texttt{new\_pubkey}, \{new\_pubkey(?x)\} \to s), \{x\} \cup \nu}$$

LET (ACCEPT)
$$\frac{c \to s, \nu}{\text{let } x = \text{accept in } c \to (+\texttt{accept}, \{accept(?x)\} \to s), \nu}$$

LET (CONNECT)
$$\frac{c \to s, \nu}{\text{let } x = \text{connect } p \text{ in } c \to (+\texttt{connect}, \{connect(p, ?x)\} \to s), \nu}$$

LET (BIND)
$$\frac{c \to s, \nu}{\text{let } x = m \text{ in } c \to (+\texttt{bind}, \{bind(?x)\} \to s[x \mapsto m]), \nu}$$

Figure 4.6: CPPL semantics

are like ours.) Additionally, the original semantics only attaches trust management formulas to strands that have already been derive, i.e., that are "true" in the theory. In contrast, our semantics produces annotated strands with untested formulas. However, since our strands do not represent execution *traces*, but execution *possibilities*, these formulas should be regarded as true when an execution actually has the behavior they represent.

Protocol Soundness. The most important ramification of the differences between the semantics is related to what proofs can be written about programs using them. Suppose we are trying to prove that a CPPL program is sound, i.e., every rely formula is supported, or true, in every execution given by the semantics.

In the original semantics, since the strands result from tested guarantees, we *can* conclude that all of the relies are true in each execution immediately. In contrast, in our semantics, the guarantees are untested, so we *cannot* conclude that all of the relies are true in each execution. However, in any run of a CPPL program that properly tests its guarantees, which is presumably all runs that are worth taking note of (i.e., are "regular", in a sense), the relies *are* true, and our semantics *can* be used to prove soundness. Moreover, the two proofs

$$\textsc{Derive}\quad \frac{sk \to s, \nu}{\text{derive } \Phi\ sk\ fk \to (\texttt{+derive}, \Phi \to s), \nu} \qquad \textsc{Derive (alt)}\quad \frac{fk \to s, \nu}{\text{derive } \Phi\ sk\ fk \to s, \nu}$$

$$\textsc{Send}\quad \frac{sk \to s, \nu}{\text{send } \Phi\ v\ m\ sk\ fk \to (\texttt{+msg}, x, m, \Phi \to s), \nu} \qquad \textsc{Send (alt)}\quad \frac{fk \to s, \nu}{\text{send } \Phi\ v\ m\ sk\ fk \to s, \nu}$$

$$\textsc{Receive}\quad \frac{sk \to s, \nu}{\text{recv } x\ m\ \Psi\ sk\ fk \to (\texttt{-msg}, x, m, \Psi \to s), \nu} \qquad \textsc{Receive (alt)}\quad \frac{fk \to s, \nu}{\text{recv } x\ m\ \Psi\ sk\ fk \to s, \nu}$$

$$\textsc{Match}\quad \frac{sk \to s, \nu}{\text{match } x\ m\ \Psi\ sk\ fk \to (\texttt{+match}, ai, x, \_ \to \texttt{-match}, ai, m, \Psi \to s), \nu} \qquad \textsc{Match (alt)}\quad \frac{fk \to s, \nu}{\text{match } x\ m\ \Psi\ sk\ fk \to s, \nu}$$

$$\textsc{Call}\quad \frac{sk \to s, \nu}{\text{call } \Phi\ x\ v^*\ u^*\ \Psi\ sk\ fk \to (\texttt{+call}, pr, x, ai, v^*, \Phi \to \texttt{-return}, ai, u^*, \Psi \to s), \{ai\} \cup \nu}$$

$$\textsc{Call (fails)}\quad \frac{fk \to s, \nu}{\text{call } \Phi\ x\ v^*\ u^*\ \Psi\ sk\ fk \to (\texttt{+call}, pr, x, ai, v^*, \Phi \to \texttt{-fail}, ai, \_ \to s), \{ai\} \cup \nu}$$

$$\textsc{Call (alt)}\quad \frac{fk \to s, \nu}{\text{call } \Phi\ x\ v^*\ u^*\ \Psi\ sk\ fk \to s, \nu} \qquad \textsc{Procedure}\quad \frac{c \to s, \nu}{\text{proc } x\ v^*\ \Psi\ c \to (\texttt{-call}, pr, x, ai, v^*, \Psi \to s), \nu}$$

Figure 4.7: CPPL semantics (cont.)

will be nearly identical, modulo this consideration.

Conclusion. In essence, these two semantics differ in their solution strategy: one records *traces* of an execution as strands and the other reports the various execution *possibilities* as strands. Intuition suggests that the two semantics agree on the meaning of protocol behavior and that the same soundness proofs can be written, but we have not yet formally shown this.

4.6 Related Work

CPPL is not our original work. However, we have greatly extended its definition and are the first to mechanically formalize it.

CPPL is uncommon amongst cryptographic protocol calculi and verifiers. Verifiers typically work with process calculi languages, such as the spi-calculus [3]. In contrast to spi, CPPL is not intended to be verified

directly; instead, it is meant to be used in implementations. Verification of CPPL specifications is through its semantic interpretation—the strand spaces model.

4.7 Conclusion

CPPL is an existing protocol calculus that is well-suited for deployment analysis: It can already be compiled to sound and cryptographically trustworthy implementations *and* it specifies one role of a protocol at a time, i.e., the part of a protocol that is executed by a single principal. This means that it captures the concerns related to deploying a single server. However, CPPL does not capture the protocol specification style found in the literature, as we discuss in the next chapter.

Chapter 5

The Test Suite

A study of cryptographic protocols and their deployments must consider real protocols. In my study I have primarily used protocols from the Security Protocols Open Repository [114] (SPORE) and from Clark and Jacob's seminal survey [24] of the protocol literature.

The following protocols were considered:

1. Amended Needham Schroeder Symmetric Key [105]
2. Andrew Secure RPC [122]
3. BAN concrete Andrew Secure RPC [15]
4. BAN modified Andrew Secure RPC [15]
5. BAN modified version of CCITT X.509 (3) [15]
6. BAN simplified version of Yahalom [15]
7. Bull's Authentication Protocol [13]
8. CAM [109]
9. CCITT X.509 (1) [20]
10. CCITT X.509 (1c) [4]
11. CCITT X.509 (3) [20]
12. Clark and Jacob modified Hwang and Chen modified SPLICE/AS [23]
13. Denning-Sacco shared key [32]
14. Diffie-Hellman [33]

15. EPMO [48]
16. GJM [41]
17. Gong [42]
18. Hwang and Chen modified SPLICE/AS [64]
19. Hwang modified version of Neumann Stubblebine [65]
20. KSL [71]
21. Kao Chow Authentication v.1 [69]
22. Kao Chow Authentication v.2 [69]
23. Kao Chow Authentication v.3 [69]
24. Kerberos V5 [107]
25. Lowe modified KSL [84]
26. Lowe modified BAN concrete Andrew Secure RPC [84]
27. Lowe modified Denning-Sacco shared key [87]
28. Lowe modified Wide Mouthed Frog [87]
29. Lowe's fixed version of Needham-Schroder Public Key [83]
30. Lowe's modified version of Yahalom [88]
31. Needham-Schroeder Symmetric Key [104]
32. Needham-Schroeder Public Key [104]
33. Neumann Stubblebine [108]
34. Otway Rees [110]
35. Paulson's strengthened version of Yahalom [113]
36. SK3 [124]
37. SPLICE/AS [144]
38. Schnorr's Protocol [28]
39. Shamir-Rivest-Adleman Three Pass Protocol [24]

40. SmartRight view-only [135]
41. TMN [130]
42. Wide Mouthed Frog [15]
43. Wired Equivalent Privacy Protocol [1]
44. Woo and Lam Pi 2 [143]
45. Woo and Lam Pi [143]
46. Woo and Lam Mutual Authentication [143]
47. Woo and Lam Pi 3 [143]
48. Woo and Lam Pi 1 [143]
49. Woo and Lam Pi f [143]
50. Yahalom [15]

The majority of these protocols are roughly categorized as authentication protocols. Many establish a new key for use between two previously unacquainted participants. Others establish a temporary session key between two acquainted participants. One in particular (EPMO) is for communicating electronic payment money-orders.

Each of these protocols have aspects that are relevant to our usage model. First, they are primarily written in a combination of stylized idiomatic specification and prose, rather than in a formal language. Second, they have participants that are useful employed with massive amounts of users. For example, many protocols have a clear 'server' that is expected to deal with many users. Third, they have participants with return from fault-tolerance. Again, there are many servers and many clients gain from not being forced to re-execute. Finally, they are not typically equipped with scalable implementations.[1]

While there is considerable overlap among goals, there is a great deal of variance in the more mechanical aspects of the protocols and the communication patterns. For example, the Andrew protocol is between two unknown participants; the Bull protocol is between three unknown participants and a trusted server; the Denning-Sacco protocol is between two unknown participants and a trust server; the EPMO is between three equally untrusted participants; etc.

Other differences abound: there are symmetric and asymmetric versions of most patterns and goals; there are instances of encrypted messages sent from, e.g., S to B via A, where A cannot decrypt the message, while in other cases S communicates directly to B, then B corresponds with A, for roughly the same effect.

Our analyses are fundamentally about how to construct and destruct messages as well as how to interpret streams of incoming messages. The protocols from the test-suite sufficiently exercise all aspects of our

[1] Naturally there are exceptions: for example, EPMO is specified in CPPL and Kerberos has very good implementations.

analysis, which conveniently allows us to experiment with our (theoretically) universally applicable analyses and theorems.

During our development, we gave no particular consideration to any protocol. Instead, we looked at protocols from an *a priori* perspective and tested our conclusions after the fact. This is good, because our work is not biased based on the kinds of protocols in the test suite—except in the case of dealing with idiomatic specifications, because we declared victory when every protocol passed. The only side-effect is that we may be wrong about *how* important various considerations are. For example, in the section on dispatching, we discuss how important, e.g., nonces are to distinguishing. These numbers may not accurately reflect the entire universe of protocols.

The most significant gap in our test suite is related to rely-guarantee proof techniques. These are the basis of the strand spaces analysis method and are essential in all of our languages and analyses. Unfortunately, they are not standard, so nearly all the protocols we consider do not include rely-guarantee formulas in their original specification or our interpretation thereof. However, our use of Coq ensures that *when present* formulas are treated appropriately; therefore, formula handling is not (and should not be!) tested empirically, but verified formula. The only appreciable place where the lack of real formulas is noticeable is that we do not reason about entailment when discussing minimal backups.

Part II

The Language

Chapter 6

The Whole Protocol Programming Language

In Part I, we discussed the formal model of cryptographic protocols we use in our work and a domain-specific language to ease specification. Before moving forward with our analyses, we must ensure that this language matches the style of protocol specification found in the literature.

Figure 6.1 shows three examples of actual protocols, as found in a state of nature. Figure 6.1 (a) is the specification of the Kerberos protocol [107]; (b) is the specification of the Kao Chow protocol from [69]; and (c) is the specification of the Yahalom protocol [15] for the SPORE repository [114].

The style of specification in each of these protocol descriptions does *not* match the style of CPPL. This entails that protocols must be rewritten. In this chapter we investigate the differences in specification style and build a language that matches the literature style.

Our example specifications contain a description of what each role of the protocol does at each step of the protocol. They say that at each step, some role a sends a message m to another role b, written $a \rightarrow b : m$. (However, it is important to understand that this is not what actually happens. In reality, a emits a message m and b receives a message m' that matches the pattern of m. Recognizing this distinction makes apparent the threat of man-in-the-middle attacks and other message mutilation in the network medium. The strand space model makes this distinction.)

All three of the protocols in Figure 6.1, and most others like them found in the literature and in repositories, have two characteristics in common:

1. They describe the entire protocol at once, whether diagrammatically (akin to a message sequence chart) or in an equivalent textual format.

2. They have certain idiomatic forms of implicit specification (as we elaborate below and in Section 6.3).

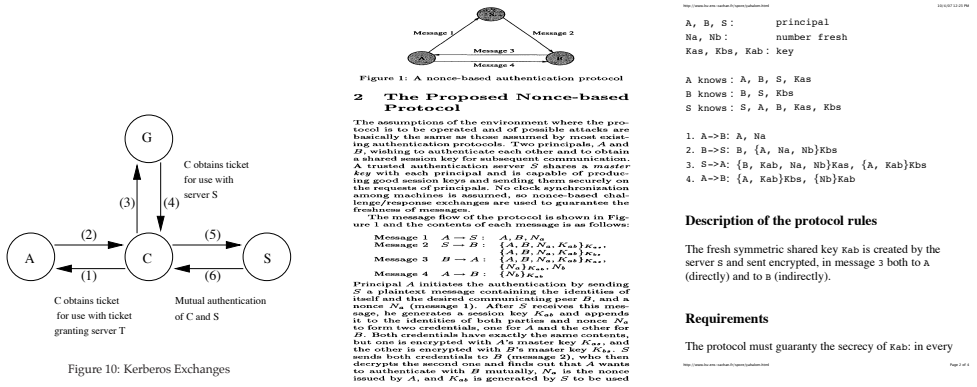


Figure 1: A nonce-based authentication protocol

2 The Proposed Nonce-based Protocol

The assumptions of the environment where the protocol is to be operated and of possible attacks are basically the same as those assumed by most existing authentication protocols. Two principals, A and B, wishing to authenticate each other and to obtain a shared session key for subsequent communication. A trusted authentication server S shares a *master key* with each principal and is capable of producing good session keys and sending them securely on the requests of principals. No clock synchronization among machines is assumed, so nonce-based challenge/response exchanges are used to guarantee the freshness of messages.

The message flow of the protocol is shown in Figure 1 and the contents of each message is as follows:

Message 1 $A \rightarrow S$: A, B, N_a
Message 2 $S \rightarrow B$: $\{A, B, N_a, K_{ab}\}_{K_{bs}}$, $\{A, B, N_a, K_{ab}\}_{K_{as}}$
Message 3 $B \rightarrow A$: $\{A, B, N_a, K_{ab}\}_{K_{as}}$, $\{N_a\}_{K_{ab}}, N_b$
Message 4 $A \rightarrow B$: $\{N_b\}_{K_{ab}}$

Principal A initiates the authentication by sending S a plaintext message containing the identities of itself and the desired communicating peer B, and a nonce N_a (message 1). After S receives this message, he generates a session key K_{ab} and appends it to the identities of both parties and nonce N_a to form two credentials, one for A and the other for B. Both credentials have exactly the same contents, but one is encrypted with A's master key K_{as}, and the other is encrypted with B's master key K_{bs}. S sends both credentials to B (message 2), who then decrypts the second one and finds out that A wants to authenticate with B mutually, N_a is the nonce issued by A, and K_{ab} is generated by S to be used

Figure 10: Kerberos Exchanges

(a) (b) (c)

Figure 6.1: Protocols in the wild

The whole-protocol nature of description is problematic because ultimately, what executes is not the "protocol" per se, but rather various software and hardware devices each implementing the individual roles. This is the impetus of CPPL's specification style. Therefore, we need a way to automatically obtain a description of a single role's behavior from this whole-protocol description. In particular, we would like to automatically generate CPPL descriptions of each role, so that we can leverage existing CPPL and strand space analyses.

The problem of obtaining individual roles from a composite description is familiar. In the realm of Web Services, is is common to *choreograph* a collection of roles by presenting a similar whole-protocol description. This has led to the definition of *end-point projection* [19], a process of obtaining descriptions of individual role behaviors from the choreography.

Unfortunately, we cannot directly lift the idea of end-point projection to the cryptographic realm because existing descriptions of end-point projection do not handle the complexities introduced by cryptography. If role A sends a message encrypted with key K, then role B can only receive the contents of that message if he has key K. Cryptography introduces information asymmetries like this into communication: it is not always the case that what one role can send, another role can receive. Existing end-point projection systems assume that such asymmetries do not exist. These systems thus focus on communication patterns and neglect communication content.

An additional problem is that protocol specifications have a few idiomatic forms. They typically do not explicitly encode: (a) what information is available at the first step of the protocol; (b) where and when various values, such as nonces[1], are generated; (c) when certain messages are *not* deconstructed; and, (d) the order in which messages components are deconstructed. We provide a whole-protocol specification language, WPPL, that allows all these to be expressed explicitly. Furthermore, we provide a transformation that removes idioms (b), (c), and (d) from a protocol that employs them.

[1]Nonces are unique random values intended to be used only once; they are used, e.g., to protect against replay attacks.

$$
\begin{array}{rcl}
s & \rightarrow & (\text{spec } rs^* \ a) \\
rs & \rightarrow & [x \ (v^*) \ (u^*)] \\
a & \rightarrow & . \quad | \quad [x \rightarrow y : \Phi \ m \ \Psi] \ a \\
 & | & \text{let } v \ @ \ x = \text{new } nt \ a \quad | \quad \text{bind } v \ @ \ x \text{ to } m \ a \\
 & | & \text{match } v \ @ \ x \text{ with } m \ a \quad | \quad \text{derive } \Phi \ @ \ x \ a \\
nt & \rightarrow & \text{nonce} \quad | \quad \text{symkey} \quad | \quad \text{pubkey}
\end{array}
$$

Figure 6.2: WPPL Syntax

Outline. We define the syntax and semantics of our language, WPPL, in Sec. 6.1. We give the end-projection from WPPL to CPPL, in Sec. 6.2. We describe our transformation from idiomatic to explicit protocol specifications in Sec. 6.3. We follow with related work, and our conclusion.

6.1 WPPL

WPPL is our domain-specific language for expressing whole cryptographic protocols with trust annotations. Like CPPL, it matches the level of abstraction of the Dolev-Yao model [35], i.e., the programmer regards the cryptographic primitives as black boxes and concentrates on the structural aspects of the protocol. In this view of protocol behavior, as each principal executes, it builds up an *environment* that binds identifiers to values encountered and compares these values with subsequent instances of their identifiers.

The Core Language. The syntax of the WPPL core language is presented in Figure 6.2. The core language has protocol specifications (s), role declarations (rs), and six types of actions (a). WPPL employs the same syntactic conventions as CPPL: Programming language identifiers are indicated by x, lists of variables by v^*, and constants (such as 42) by k. The language has syntax for trust management formulas—by convention we write guaranteed formulas as Φ and relied formulas as Ψ.

Examples. Figure 6.3 shows the Yahalom protocol [15] rendered as a WPPL specification. It matches the construction commonly used in the literature, but is slightly more verbose. Figure 6.4 shows the Kao Chow [69] protocol as an idiomatic WPPL specification. We will discuss what exactly is idiomatic in this specification in Sec. 6.3.

Syntactic Conventions. m refers to both messages and message patterns because of the network model. Consider the m expression `(a, b, na, kab)`. The sender looks up `a`, `b`, etc., in its environment to construct a message that it transmits. From the receiver's perspective, this is a pattern that it matches against a received message (and binds the newly-matched identifiers in its own environment). Because of the intervening network, we cannot assume that the components bound by the receiver are those sent by the sender.

A protocol specification declares roles and an action. Role declarations give each role a name x, a list v^*

```
 1 (spec ([a (a b s kas) (kab)]
 2        [b (a b s kbs) (kab)] [s (a b s kas kbs) ()])
 3  let na = new nonce
 4  [a -> b : a, na]
 5  let nb = new nonce
 6  [b -> s : b, {|a, na, nb|} kbs]
 7  let kab = new symkey
 8  bind stob to {|a, kab|} kbs
 9  [s -> a : {|b, kab, na, nb} kas, stob]
10  bind atob to {|nb|} kab
11  [a -> b : stob, atob]
12  match stob with {|a, kab|} kbs
13  match atob with {|nb|} kab .)
```

Figure 6.3: Yahalom in WPPL

```
1 (spec ([a (a b s kas) (kab)]
2        [b (b s kbs) (kab)] [s (a b s kas kbs) ()])
3  [a -> s : a, b, na:nonce]
4  [s -> b : {|a, b, na, kab|} kas, {|a, b, na, kab|} kbs]
5  [b -> a : {|a, b, na, kab|} kas, {|na|} kab, nb:nonce]
6  [a -> b : {|nb|} kab] .)
```

Figure 6.4: Kao Chow in WPPL

of formal parameters, and a list u^* of identifiers that will be returned by the protocol representing the "goal" of the protocol. Actions are written in continuation-passing style. Although the grammar requires types attached to every identifier, we use a simple type-inference algorithm to alleviate this requirement. (Similarly, we infer the principal executing each action.) However, these technical details are standard, so we do not elaborate them.

Informal Execution Semantics. The execution of a **return** (.) action returns the return identifiers (u^*) from each role. On a **comm** ($\rightarrow$) action, x derives the formulas Φ and sends the message m and y receives pattern m, relies on the formulas Ψ, and executes a; if y cannot match m, then y fails. The execution of a **new** (let) action generates a new value, binds it to v on x, and executes a on x. The execution of a **bind** action binds message m to v on x and executes a on x. The execution of a **match** action attempts to match m against v on x, if successful it expands the environment on x and executes a on x; if unsuccessful, then x fails. The execution of a **derive** action attempts to derive the formulas Φ on x; if successful a executes on x, otherwise, x fails. If any role fails, its failure is isolated in the sense that other roles do not automatically fail, but may fail if they depend on failed role in any way.

Semantic Interpretation. Like CPPL, the WPPL semantics identifies a set of *strands* [131] annotated with trust formulas as the meaning of each role in a protocol.

6.1.1 Well-formedness

A WPPL specification is well-formed when, for each declared role, the identifiers it uses are bound and the messages it sends or receives are well-formed. For example, the Yahalom protocol is well-formed, while Kao Chow is not: for instance, on line 3, `na` is not bound.

We write $\kappa, \sigma \vdash^r_\rho a$ to mean that an action a is well-formed for role r, that r expects to return ρ, in the environment σ, when it has previously communicated with the roles in κ, where κ and σ are sets of identifiers, r is an identifier, ρ is a list of variables, and a is an action. (See Figure 6.5.) We write $\vdash^r s$ to mean that the specification s is well-formed for the role r and $\vdash s$ to mean that specification s is well-formed for all roles r that appear therein. (See Figure 6.6.)

The parameter ρ is necessary because **return**s do not specify what is being returned. The parameter κ is necessary because we do not require explicit communication channels. At first glance, it may seem that we must only ensure that the name of a communication partner is in σ, but this is too conservative. We are able to reply to someone even if we do not know their name. Therefore, κ records past communication partners.

We choose to require well-formed WPPL specifications to be causally connected [19]. This means that the actor in each action is the same as that of the previous action, except in the case of communication. Our presentation does not rely on this property, so we believe we could elide it. However, we believe that to do so would allow confusing specifications that are not commonly found in the literature.

6.1.2 Semantics

The semantics of WPPL contains an implicit end-point projection. Each phrase is interpreted separately for every role as a set of strands that describes one possible local run of that role. These sets are derived from the many possible strands that may be derived as descriptions for particular phrases, as in CPPL.

We write $a \rightarrow^r_{\rho,\kappa} s, \nu$ to mean that an action a, when executed by role r, with return variables ρ, and the current connections κ may produce strand s and requires ν to all be unique identifiers.

We use a few fruitful strategies in this semantics. First, to ensure that the same identifiers are bound in the strand, we send messages ("internal dialogue") that contain no information, but where the formulas bind identifiers. Second, the interpretation of matching is to emit an identifier, then "over-hear" it, using a message pattern. Third, when binding an identifier, the identifier is replaced by the binding in the rest of the strand.

The semantics is given in Figure 6.7 and is included in the Coq formalization as the inductive definition of `eval_action` (17 rules) and `eval_spec` (2 rules).

We write $sp \rightarrow^r s, \nu$ to mean that spec sp, when executing role r may produce strand s and requires ν to all be unique identifiers. We prove that this semantics always results in well-formed strands for well-formed specifications.

$$\textsc{Return}\ \frac{\rho \subseteq \sigma}{\kappa, \sigma \vdash^r_\rho .} \qquad \textsc{Comm (Send)}\ \frac{t \in \kappa \vee t \in \sigma \qquad \sigma \vdash \Phi \qquad \sigma \cup \mathit{bound}(\Phi) \vdash_s m \qquad \kappa \cup \{t\}, \sigma \cup \mathit{bound}(\Phi) \vdash^r_\rho a}{\kappa, \sigma \vdash^r_\rho [r \to t : \Phi\ m\ \Psi]\ a}$$

$$\textsc{Comm (Receive)}\ \frac{\sigma \vdash_r m \qquad \sigma \cup \mathit{bound}(m) \vdash \Psi \qquad \kappa \cup \{f\}, \sigma \cup \mathit{bound}(m) \cup \mathit{bound}(\Phi) \vdash^r_\rho a}{\kappa, \sigma \vdash^r_\rho [f \to r : \Phi\ m\ \Psi]\ a}$$

$$\textsc{Comm (Skip)}\ \frac{r \neq f \qquad r \neq t \qquad \kappa, \sigma \vdash^r_\rho a}{[f \to t : \Phi\ m\ \Psi]\ a} \qquad \textsc{New}\ \frac{\kappa, \sigma \cup \{v\} \vdash^r_\rho a}{\kappa, \sigma \vdash^r_\rho \text{let } (v{:}nt)\ @\ r = \text{new } nt\ a} \qquad \textsc{New (Skip)}\ \frac{r \neq x \qquad \kappa, \sigma \vdash^r_\rho a}{\kappa, \sigma \vdash^r_\rho \text{let } v\ @\ x = \text{new } nt\ a}$$

$$\textsc{Bind}\ \frac{\sigma \vdash_s m \qquad \kappa, \sigma \cup \{v\} \vdash^r_\rho a}{\kappa, \sigma \vdash^r_\rho \text{bind } (v{:}\text{msg})\ @\ r \text{ to } m\ a} \qquad \textsc{Bind (Skip)}\ \frac{r \neq x \qquad \kappa, \sigma \vdash^r_\rho a}{\kappa, \sigma \vdash^r_\rho \text{bind } v\ @\ x \text{ to } m\ a}$$

$$\textsc{Match}\ \frac{v \in \sigma \qquad \sigma \vdash_r m \qquad \kappa, \sigma \cup \mathit{bound}(m) \vdash^r_\rho a}{\kappa, \sigma \vdash^r_\rho \text{match } (v{:}\text{msg})\ @\ r \text{ with } m\ a} \qquad \textsc{Match (Skip)}\ \frac{r \neq x \qquad \kappa, \sigma \vdash^r_\rho a}{\kappa, \sigma \vdash^r_\rho \text{match } v\ @\ x \text{ with } m\ a}$$

$$\textsc{Derive}\ \frac{\sigma \vdash \Phi \qquad \kappa, \sigma \cup \mathit{bound}(\Phi) \vdash^r_\rho a}{\kappa, \sigma \vdash^r_\rho \text{derive } \Phi\ @\ r\ a} \qquad \textsc{Derive (Skip)}\ \frac{r \neq x \qquad \kappa, \sigma \vdash^r_\rho a}{\kappa, \sigma \vdash^r_\rho \text{derive } \Phi\ @\ x\ a}$$

Figure 6.5: WPPL action well-formedness

Theorem 2. *(`eval_spec_type`) If* $\vdash^r sp$ *and* $sp \to^r s, \nu$ *then* $\vdash s$.

Proof summary. We prove this via the correctness of compilation into CPPL and using CPPL's similar theorem. We apply Theorems 1 and 6. □

Adversary. Because the semantics of a WPPL specification is a set of strands, the Dolev-Yao adversary of the general strand model is necessarily the adversary for WPPL. A Dolev-Yao adversary has the capability to create, intercept, and redirect messages. He can redirect messages from any party to any party. He can intercept messages to learn new information. He can create messages based on previously learned information. His only constraint is that he cannot break the cryptographic primitives, i.e., he must possess the appropriate key to decrypt a message. Therefore, the only information he knows is derived from an insecure context.

However, there are two additional points. First, the WPPL semantics uses "internal dialogue", like CPPL, to encode some behavior, such as pattern matching and generating fresh values. Second, WPPL contains a syntactic style that is dangerous because it obscures what the adversary is capable of. When we write `[a -> b : {|nb|} kab]` on line 6 of the Kao Chow protocol, it appears that we are writing "`a` sends to `b` ...", i.e., that `a` is communicating with `b`. This is very misleading, because it obscures potential middle-men

$$\textsc{Role}\ \frac{\emptyset, v^* \vdash^r_{u^*} a}{\vdash^r \text{spec}\ (\ldots, [r\ (v^*)\ (u^*)], \ldots)} \qquad \textsc{Specification}\ \frac{\forall r \qquad [r\ (v^*)\ (u^*)] \in rs^* \qquad \vdash^r \text{spec}\ rs^*\ a}{\vdash \text{spec}\ rs^*\ a}$$

Figure 6.6: WPPL specification well-formedness

and message redirection. We could propose to write `[a -> {|nb|} kab -> b]` as an alternative, but this is only a marginal improvement and is not the style used by protocol designers.

6.2 End-Point Projection

Our end-point projection of WPPL into CPPL is realized as a compiler. We will write $a \Rightarrow^r_{\rho,\kappa} c$ to mean that the projection of a for the role r, whose return variables are ρ, when κ are all roles r has communicated with, is c. The definition is given in Figure 6.8.

The compilation is straight-forward, by design of WPPL, and is, in principle, no different that previous work on this topic. The only interesting aspect is that CPPL uses channels in communication, while WPPL uses names. Thus, we must open channels as they are necessary. This is accomplished by the auxiliaries `let_connect` and `let_accept`, which produce a **let** statement that opens the channel through the appropriate means. These introduce new bindings in the CPPL procedure that are not in the WPPL specifications, and must be specially tracked. We prove that both well-formedness and meaning are preserved:

Theorem 3. *(`compile_action_type`) If* $\kappa, \sigma \vdash^r_\rho a$ *and* $a \Rightarrow^r_{\rho,\kappa} c$ *then* $\sigma \vdash c$.

Proof summary. We proceed by induction on the action, a, followed by lemmas about channel creations and set theory solvers. □

Theorem 4. *(`compile_action_correct`) If* $a \Rightarrow^r_{\rho,\kappa} c$, *then if* $a \rightarrow^r_{\rho,\kappa} s, \nu$ *then* $c \rightarrow s, \nu$. *(Notice that the semantic interpretation* s, ν *is identical.)*

Proof summary and discussion. We proceed by induction on the action, a, inversion on the evaluation judgment, then manual description of which CPPL evaluation rules apply in each case. This proof could probably be automated, but I wasn't as skilled at Coqetry when I wrote it.

This theorem expresses preservation of *semantic meaning*: the strand s and the unique set ν are identical in both evaluation judgments. This means that the CPPL phrase *perfectly* captures the meaning of the WPPL phrase. □

Another two proofs about the compiler show that there is always a **return** statement and that all **return** statements include the same guarantees and returned variables. (Recall that this is a well-formedness condition of CPPL procedures: Sec. 4.3.)

We lift the compiler of actions to specifications. We write $s \Rightarrow^r p$ to mean that the projection of the specification s for the role r is the procedure p. For some roles, namely those that are not declared in the specification, we will write $s \Rightarrow^r \bot$ to indicate the lack of a projection for the role.

Theorem 5. *(`compile_spec_type`)* $\vdash^r s$ *implies that there is a* p *such that* $s \Rightarrow^r p$ *and* $\vdash p$.

Proof summary. This is a simple application of Theorem 3. □

Theorem 6. *(`compile_spec_correct`)* $sp \rightarrow^r s, \nu$ *implies* $sp \Rightarrow^r p$ *implies* $p \rightarrow s, \nu$.

Proof summary. This is a simple application of Theorem 4 and the CPPL procedure evaluation rule. □

Example. Figure 6.9 is the compilation of one role of Yahalom. The others are similar.

6.3 Explicit Transformation

Having shown a correct end-point projection, we turn to the problem of handling the idioms in specification. For convenience, we reproduce Kao Chow in Figure 6.10

This specification is not well-formed because:

1. `a` cannot construct the message on line 3 because `na` is not bound.
2. `s` cannot construct the message on line 4 because `kab` is not bound.
3. `b` cannot match the message on line 4 because `kas` is not bound.
4. `b` cannot construct the message on line 5 because `nb` is not bound.
5. `a` cannot match the message on line 5 because `kab` is not bound.

We are specifically interested in allowing protocols to be taken from standard presentations in the literature and used with our compiler. As other researchers have noted [2, 9], protocols like this often make use of very loose constructions and leave many essentials implicit. One approach is to reject such protocols outright and force conformance. Our approach is to recognize that there is a de facto idiomatic language in use and support it, rather than throwing out the large number of extant specifications.

The Kao Chow protocol contains all the idioms that we most commonly encounter in the literature:

I. Implicitly generating fresh nonces and keys by using a name that does not appear in the rest of the specification (e.g., `na`, `kab`, and `nb`).

II. Allowing roles to serve as carriers for messages that they cannot inspect, without indicating this (e.g., the first part of line 4's message).

III. Leaving the order of pattern-matching unspecified (e.g., line 5).

Our transformation handles all these implicit specifications, resulting in a version of this protocol that is well-formed.

Overview. Our transformation has three stages. The first removes idiom I, by generating new bindings for nonces and keys; it is discussed in Sec. 6.3.1. The second removes idiom II and is the first step of removing idiom III. It lifts out message components that are possibly unmatchable, i.e., encrypted or hashed, and binds them to identifiers. This is discussed in Sec. 6.3.2. The third stage matches bound identifiers against their construction pattern, when this will succeed. This sequences pattern matching, removing idiom III, and recovers any losses temporarily created by stage two. It is discussed in Sec. 6.3.3. We conclude with an evaluation of these transformations (Sec. 6.3.4).

6.3.1 Generation

Our first transformation addresses problems 1, 2 and 4 above by explicitly generating fresh values for all nonces, symmetric keys, and public keys that appear free in the entire protocol. After transformation, Kao Chow is as in Figure 6.11.

This transformation is very simple, so we do not present it in detail. Instead, we explain its correctness theorem. We write *gen*(*s*) as the result of this stage for specification *s*. Our theorem establishes a condition for when the first idiom is removed:

Theorem 7. *(`gen_spec_correct`) For all s, if its action does not contain an instance of recursive binding, then for all vi, if vi appears free in gen*(*s*)*, then there exists a type vt, such that vt is* not `nonce`*,* `symkey`*, or* `pubkey` *and vi appears free in s with type vt.*

Proof summary. Induction on the structure of the action embedded in the specification *s*, followed by lemmas about when an identifier and type can be freshly generated. □

An action has recursive binding if it contains as a sub-action `bind` *r v m a* and *v* appears in *m*. These actions are not strictly well-formed, because *v* is not bound in *m*. However, we cannot assume that input specifications are well-formed—our transformation is to make them so! So, we have parceled out the smallest amount of well-formedness necessary.

This theorem says that any free identifier (a) is not of one of the types for which we can construct a fresh value and (b) is free in the original specification, i.e., was *not* introduced by our transformation. Thus, the first idiom is successfully removed.

Though we have removed idiom I, we still have problems 3 and 5, so our modified Kao Chow protocol is still not well-formed.

6.3.2 Lifting

The second stage transforms each message construction by introducing a message binding for each message component. It binds those components that can potentially fail to match, namely signing and encryption (which require the key to be matched), and hashing (which requires the hash contents to be matched). This results in Kao Chow further being rewritten to the form shown in Figure 6.12.

As a result of this transformation, `b` can transmit `msg0` without needing to inspect it on line 4.

This serves to ensure (a) all matching sides of communication are well-formed; (b) messages that are carried without inspection are well-formed for sending; and, (c) sequencing is completely unspecified on the receiving side. We prove the following theorem about this stage that establishes criterion (a):

Theorem 8. *(`bindc_spec_type_all`) For all s and roles in s, the specification resulting from the lifting stage applied to s satisfies criterion (a).*

Proof summary. Induction on the list of role declarations in *s* followed by induction on the integer parameter of the transformation. There are a few instances of ensuring that this parameter is always decreasing, but other than this the proof goes through easily by definition of the transformation. □

We argue criteria (b) and (c) informally.

One interesting part of our transformation is that it is not structurally recursive in the action. In the cases for communication, binding, and matching, the translation is recursively applied to the result of replacing all instances of the lifted message components in the continuation. Instead, we recur on a natural number bound. We prove that the depth of the syntax of the action is a sufficient bound for the correctness of this function.

Though we have removed idiom II, we still have problem 6, and we have introduced many other instances of idiom III.

6.3.3 Opening

The third stage introduces pattern-matching at each point where a message previously bound may be successfully matched against the pattern it was bound to. This results in Kao Chow being rewritten to the final, well-formed, form shown in Figure 6.13.

After this transformation, every message that can possibly be deconstructed by each role is deconstructed. This removes idiom III by fully specifying the order of pattern matching. In particular, it thereby removes instances of idiom III introduced by the second stage.

Although it follows from this stage's mission statement, it is not necessarily intuitive that this stage will also check that previously unverified message contents are correct. Since this pattern occurs when a message that *could not* be deconstructed becomes transparent, this pattern *is* handled by this step. For example, if a commitment message, $hash(m)$, has been received, but the contents, m, is unknown until a later step; this stage will check the commitment at the appropriate time.

This stage must solve the following problem: find some order whereby the messages may be matched, or "opened." A message may be opened if (a) the identifier it is bound to is bound (which isn't necessarily the case: e.g., `msg0` is not bound on line 7) and (b) it is well-formed for receiving in the environment. At each line of the specification, our transformation will compute the set of messages that may be opened, and the order to open them in. If a message cannot be opened, it will be reconsidered in subsequent lines. Note that this is a recursive set, because opening a message extends the environment, potentially enabling more messages to be opened. Thus, messages that can't be opened may become amenable to opening after some other message is opened.

The core of the transformation is a function that computes this set for each line. This function, `open_bmsgs`, in principle accepts a B, list of identifiers and message patterns, i.e., the bound messages, and σ, an environment. It partitions B into two lists: C, the bound messages that *cannot* be opened; and O, those that can.

In fact, this function cannot not only receive these two values as arguments, because it cannot be defined recursively on either of them. Instead, it is supplied an additional integer bound. We prove that if the bound is greater than the number of messages, our properties hold.

Theorem 9. *(`open_bmsgs_closed`) If* `open_bmsgs` $B\ \sigma = (C, O)$*, then if* (i, m) *is in* C*, then either* $i \notin \sigma \cup ids(O)$ *or* $\sigma \cup ids(O) \nvdash_r m$*.*

Proof summary. Induction on the length of B followed by equational theorems about the standard list functions (partition and length) used in the definition. The most interesting part is ensuring that we can always detect when partition doesn't decrease the size of the active list and it is safe to stop. □

This first correctness theorem shows that the function is correct in declaring messages as non-openable. The second theorem shows that the function is correct in marking message as openable.

Theorem 10. *(`open_bmsgs_opened`) If* `open_bmsgs` $B\ \sigma = (C, O)$*, then if* $O = p\ @\ (i, m) :: q$*, then* $i \in \sigma \cup ids(p)$ *and* $\sigma \cup ids(p) \vdash_r m$*.*

Proof summary. The structure of this proof is almost identical to Theorem 9, although the details are obviously different. □

Our transformation is trivial, given `open_bmsgs`. In essence, it keeps track of the environment of the role being transformed and the bound messages. Then, it calls `open_bmsgs` and uses the result of a small auxiliary, `build_match`, that actually destructures bound messages, thereby removing idiom III. We prove the following theorem about `build_match`:

Theorem 11. *(`build_match_type`) If* `open_bmsgs` $B\ \sigma = (C, O)$*, then if* $\kappa, \sigma \cup ids(O) \vdash^r_\rho a$ *and* `build_match` $r\ O\ a = a'$ *then* $\kappa, \sigma \vdash^r_\rho a'$*.*

Proof summary. Application of Theorem 10 followed by a very simple induction of the length of O. □

6.3.4 Evaluation

We now evaluate the effectiveness of these transformations. We did not "evaluate" end-point projection because our theorem established that it succeeds for *all* inputs. Similarly, we have established appropriate correctness conditions for each of the three transformations. However, we still need to determine how well the three transformations actually cover the space of protocols found in practice. (Note that we cannot argue the correctness of the composition of the three transformations, because their input is not well-formed, so there is no foundation for their "correctness".)

We attempted to encode fifty of the protocols in the Security Protocols Open Repository [114] (SPORE) in WPPL to evaluate the language and the transformations.

We successfully encoded forty-three of the protocols. We consider this compelling evidence that WPPL is **useful for producing protocols**. Of these protocols, *zero* were well-formed as they appear in the repository and *all* are well-formed after applying the composed transformation. This demonstrates the transformations are effective at explicating protocol specifications—further enhancing the utility of WPPL.

Weaknesses and Limitations. WPPL has a few weaknesses that prevent all protocols to be encoded.

First, WPPL cannot express unique cryptographic primitives. It can only express asymmetric or symmetric modes of encryption or signing and hashing. Therefore, it cannot express protocols that build these (and other) primitives. For example, the Diffie-Hellman protocol [33] develops a method of creating shared symmetric keys. We cannot guarantee from within the framework of WPPL that this theory is correct, nor provide the axioms, and guarantee that the generated keys are really symmetric. In essence, this protocol is too low-level for our theory.[2]

Second, there is no way to express conditional execution in WPPL. Thus, there is only one path through a protocol and protocols with built-in failure handling, such as the GJM protocol [41], cannot be written. We could have extended our work to allow the expression of these protocols, but WPPL attempts to capture the spirit of traditional protocol description, and we cannot identify a community consensus on how to write conditional executions.

Third, protocols that rely on parallel execution are not well-formed, as mentioned earlier. For example, the following would be valid WPPL syntax:

```
1 A -> B : m1
2 A -> B : m2
```

But, it is not well-formed, because on line 2, A is the actor instead of B. The idiom removal transformation is incapable of removing this instance of parallelism.

[2] However, there is a back door through the trust management logic. This logic could be extended with any necessary computation predicates to support whatever operates are needed to formalize the protocol. This represents a considerable task, and it would be inappropriate to assert that we support every possible protocol theory through these means.

6.4 Related Work

End-point Projection. Carbone, Honda, and Yoshida [19] have presented an end-point projection theory for the Choreography Description Language. We are inspired by their presentation, but we differ on a few fundamental points. First, in our analysis model there is an adversary, and therefore we must reason about the effectiveness of cryptographic methods and trust relations. In contrast, they assume all participants are honest and cooperative. By not explicitly handling cryptography, they do not have to specify a well-formedness condition to identify information asymmetries between protocol participants as we do in Sec. 6.1.1. Their well-formedness conditions are entirely related to session usage, because there is no information asymmetries between participants. Additionally, because they work in the Web Services space, sessions are an essential part of their work, whereas in the cryptographic space, sessions are not assumed to exist. In fact, many cryptographic protocols serve to establish some particular notion of a session. Therefore, we do not need the detailed well-formedness conditions they impose on correct session usage. Finally, they allow conditional and parallel protocol execution, which we do not allow. We found very few protocols in our literature that used this features. They would *not* be complicated to add, but it would be difficult to find a format that matches the preferences of protocol designers (because they have not been revealed.)

Sabri and Khedri [121] employ the Carbone framework of end-point projection in their development of a framework for capturing the knowledge of protocol participants. They explicitly acknowledge (p. 33) that it is necessary to verify the well-formedness of protocols given information asymmetries, but do not consider this problem in their work. Rather, they assume they are resolved appropriately. Therefore, our work is complementary to theirs, as it provides the necessary machinery to integrate cryptography into a global calculus end-point projection.

Corin et al. [26] perform end-point projection in their work on session types. A session type is a graph that expresses a global relation among protocol participants, including the pattern of legal communications. When implementing a role, it is necessary to consider that role's view of the graph. End-point projection is used to derive these session views as types that can be verified at each end-point. Corin's application of end-point projection is different from either Carbone's or ours. While Corin uses EPP at the type-level, we use it at the program-level. Nevertheless, the technique is fundamentally the same.

Program Slicing. The essence of end-point projection is program slicing [140]. Program slicing is a technique to take a program and remove parts of it that are irrelevant to some particular party. For example, the program slice with respect to some variable is the subset of that program which influences the value of the variable. In our case, we will be slicing a program with respect to some participant and removing parts of the program that do not concern that party. In particular, if the role in question receives a message, it need not be concerned with the generation of that message, other than for the purpose of the types, etc.

Based on program slicing, there has been some work on the generation of a client and a server from a single program [106]. In this work some particular values and functions are said to belong to the client or

the server, and the analysis and transformation separates the two programs and generates the communication necessary to make the client values and functions available to the server when necessary, and vice versa. There are many similar results related to compilation of client and server parts of Web applications [145, 25, 123, 103].

Compiling Traditional Protocol Specifications. Although not typically presented as end-point projection, there are a number of research projects that tackle this problem in some form. For example, CAPSL [99] is a system that transforms a protocol into runnable Java code. Their specifications describe the entire protocol, so their compiler counts as an end-point projection. Our work is distinct for multiple reasons. First, the CAPSL specifications do not support the rely-guarantee model, like we do, as analyzed in the Strand Spaces model. Instead, CAPSL uses a particular language of assumptions, axioms, protocol goals, etc., specifically targeted for the NRL Protocol Analyzer [129, 96]. Second, they require programs to be fully explicit and annotated to resolve idioms. This, in turn, means that protocols do not resemble the style used by actual practitioners, and therefore CAPSL does not require an idiom removal transformation as we do.

A system related to CAPSL is Casper [85], "a compiler for the analysis of security protocols". This compiler takes programs written in a whole-protocol form and transforms it into CSP processes that can be checked with the FDR model-checker [119]. This system is intended not only to specify the protocol, but also the security properties *about* that protocol. Like CAPSL, these properties are not expressed in the Strand Spaces model and protocols are required to be fully explicit and devoid of idiomatic conventions. Other than this, however, Casper addresses more issues involved in protocol development than WPPL, because it deals with properties. This further indicates the difference between protocol development and protocol deployment, WPPL's focus.

Both CAPSL and Casper use the `%` operator to indicates when some value cannot be understood by the receiver and should be treated as a black box. A similar approach is taken by AVISS [6] and CVS [37]. This is a way of annotating where an information asymmetry exists. Such annotations are essentially generated by our idiom removal process.

A similar system to Casper was developed by Jacquemard et al. [66]. This work compiles whole-protocol specifications into rewrite rules that model the protocol and can be verified with the daTac theorem-prover [136]. Like Casper, this system specifies the protocol as well as properties about it. The main advantage over Casper is that daTac can be used to handle infinite state models. In this work they do deal with information asymmetries, in a way, by tracking the knowledge available to each principal at each point of the protocol. However, they do not revisit earlier message components that were treated as black boxes when the knowledge needed to inspect their contents becomes available as we do. So, the following protocol would not be correct:

```
1 [A -> B : { m } k]
...
2 [A -> B : k ]
```

```
3 [B -> A : m ]
```

While in WPPL it would, because the idiom remover would notice that after step 2, the message received in step 1 may be parsed. Additionally, daTac's process of dealing with idioms and information asymmetries is deeply and implicitly embedded in their semantics, rather than an orthogonal step (as we present it).

Explication. Each system that attempts to employ traditional protocol specifications must address the idioms we have discussed. Additionally, these idioms have been mentioned, in part, in other work. For example, Bodei et al. [9, 8] mention the idioms of traditional protocol specifications. In their work they give some advice for how to manually make such specifications explicit, but they do not automate this process. Among their advice is to add the source and destination names to the message sent: to transform `[a -> b : m]` into `[a -> b : a, b, m']`. Their intent is to simulate the underlying network headers that include source and destination. However, we believe that this does not represent a true idiom used in the traditional protocol literature. It is common to see protocol descriptions that contain messages such as `[a -> b : a, m]`. If the author intended the names to be implicitly sent, then adding the `a` here would not be useful. However, this is not a significant aspect of either our work or their work and we could easily adopt their advice.

Briais and Nestmann [12] also investigate the formal semantics of traditional protocol specifications. They try to address the three of the four forms of informality mentioned by Abadi [2]. Of these four forms, only one corresponds to parts of our transformation or specification language. The first task is to "make explicit what is known before a protocol is run", which we require in the WPPL specification, and "what is to be generated freshly during a protocol run", which we detect and remove on the behalf of the protocol designer as step 1 of our idiom-removing transformation. Like our work, they do not require the usage of the `%` operator and instead only require principals to understand messages as much as they can upon reception. However, they do not revisit old messages, as we do.

Caleiro et al. [16, 17] study the semantics of traditional protocol specifications. In their work, they focus on the internal actions of principals. They give a manual strategy for encoding traditional whole-protocol specifications into a number of single-role specifications in their semantic framework. Their denotational semantics of these specifications makes explicit when and how incoming messages are checked and outgoing messages generated. They also provide a transformation of their specifications into a variant of the spi-calculus [54]. They prove that this transformation is meaningfully related to the denotational semantics. In contrast, our work takes traditional specifications mostly as-is and directly provides a semantics in the strand spaces model. We also provide a formally verified compilation to CPPL for deployment purposes. An important similarity in the two works is that in the their approach messages that cannot be understood are represented as variables in their "incremental symbolic runs", while in our approach, the idiom removal transformation introduces these variables directly into the specification and checked when possible.

6.5 Future Work

In the future, we would like to extend WPPL to support conditional and parallel execution, and better integrate our work with the existing results from Carbone, et al., for Web Services [19].

6.6 Conclusion

We have presented WPPL, a programming language for whole-protocol specification, along with an end-point projection from WPPL to CPPL. We have shown that this projection is correct through verified proofs. We have also given a transformation that resolves idioms in traditional protocol presentations. We have shown properties of this transformation with verified proofs. We have validated our transformations by applying them successfully to eight-six percent of the protocols in the SPORE repository [114].

This shows that WPPL can capture the specification style used by protocol designers, and therefore, analyses on WPPL are most usable that those on other languages. However, we will not define analyses over WPPL. Instead, we rely on the end-point projection of WPPL into CPPL and investigate protocols specified in CPPL. Since the projection is meaning preserving, the analysis results are compatible with the WPPL specification.

Note. A paper based on the content of this chapter appeared at the European Symposium on Research in Computer Security in 2008.

FAIL
$$a \to^{r}_{\rho,\kappa} (+\texttt{fail}, ai, \_), \emptyset$$

RETURN
$$. \to^{r}_{\rho,\kappa} (+\texttt{return}, ai, \rho, \_), \emptyset$$

NEW
$$\frac{a \to^{r}_{\rho,\kappa} s, \nu}{\text{let } x \ @ \ r = \text{new } nt \ a \to^{r}_{\rho,\kappa} (+\texttt{new\_}nt, \{new\_nt(?x)\} \to s), \{x\} \cup \nu}$$

NEW (NEXT)
$$\frac{ar \neq r \qquad a \to^{r}_{\rho,\kappa} s, \nu}{\text{let } x \ @ \ ar = \text{new } nt \ a \to^{r}_{\rho,\kappa} s, \nu}$$

BIND
$$\frac{a \to^{r}_{\rho,\kappa} s, \nu}{\text{bind } x \ @ \ r \text{ to } m \ a \to^{r}_{\rho,\kappa} (+\texttt{bind}, \{bind(?x)\} \to s[x \mapsto m]), \nu}$$

BIND (NEXT)
$$\frac{ar \neq r \qquad a \to^{r}_{\rho,\kappa} s, \nu}{\text{bind } x \ @ \ ar \text{ to } m \ a \to^{r}_{\rho,\kappa} s, \nu}$$

DERIVE
$$\frac{a \to^{r}_{\rho,\kappa} s, \nu}{\text{derive } \Phi \ @ \ x \ a \to^{r}_{\rho,\kappa} (+\texttt{derive}, \Phi \to s), \nu}$$

DERIVE (NEXT)
$$\frac{ar \neq r \qquad a \to^{r}_{\rho,\kappa} s, \nu}{\text{derive } \Phi \ @ \ ar \ a \to^{r}_{\rho,\kappa} s, \nu}$$

COMM (SEND,CONNECTED)
$$\frac{tr \in \kappa \qquad a \to^{r}_{\rho,\kappa} s, \nu}{[r \to tr : \Phi \ m \ \Psi] \ a \to^{r}_{\rho,\kappa} (+\texttt{msg}, tr\_\texttt{chan}, m, \Phi \to s), \nu}$$

COMM (SEND,CONNECT)
$$\frac{tr \notin \kappa \qquad a \to^{r}_{\rho,\kappa \cup \{tr\}} s, \nu}{[r \to tr : \Phi \ m \ \Psi] \ a \to^{r}_{\rho,\kappa} (+\texttt{connect}, \{connect(tr, ?tr\_chan)\} \to +\texttt{msg}, tr\_\texttt{chan}, m, \Phi \to s), \nu}$$

COMM (RECV,CONNECTED)
$$\frac{fr \in \kappa \qquad a \to^{r}_{\rho,\kappa} s, \nu}{[fr \to r : \Phi \ m \ \Psi] \ a \to^{r}_{\rho,\kappa} (-\texttt{msg}, fr\_\texttt{chan}, m, \Phi \to s), \nu}$$

COMM (RECV,ACCEPTED)
$$\frac{fr \notin \kappa \qquad a \to^{r}_{\rho,\kappa \cup \{fr\}} s, \nu}{[fr \to r : \Phi \ m \ \Psi] \ a \to^{r}_{\rho,\kappa} (+\texttt{accept}, \{accept(?fr\_chan)\} \to -\texttt{msg}, fr\_\texttt{chan}, m, \Psi \to s), \nu}$$

COMM (NEXT)
$$\frac{tr \neq r \qquad fr \neq r \qquad a \to^{r}_{\rho,\kappa} s, \nu}{[fr \to tr : \Phi \ m \ \Psi] \ a \to^{r}_{\rho,\kappa} s, \nu}$$

MATCH
$$\frac{a \to^{r}_{\rho,\kappa} s, \nu}{\text{match } x \ @ \ r \text{ with } m \ a \to^{r}_{\rho,\kappa} (+\texttt{match}, ai, x, \_ \to -\texttt{match}, ai, m, \Psi \to s), \nu}$$

MATCH (ALT)
$$\frac{ar \neq r \qquad a \to^{r}_{\rho,\kappa} s, \nu}{\text{match } x \ @ \ ar \text{ with } m \ a \to^{r}_{\rho,\kappa} s, \nu}$$

SPECIFICATION
$$\frac{[r \ (v^*) \ (u^*)] \in rs^* \qquad a \to^{r}_{u^*,\emptyset} s, \nu}{\text{spec } rs^* \ a \to^{r} (-\texttt{call}, pr, r, ai, v^*, \_ \to s), \nu}$$

Figure 6.7: WPPL semantics

$$\textsc{Return}\quad \frac{}{. \Rightarrow^r_{\rho,\kappa} \text{return } \_\ \rho}$$

$$\textsc{Comm (Send, Connected)}\quad \frac{t \in \kappa \qquad a \Rightarrow^r_{\rho,\kappa} c'}{[r \to t : \Phi\ m\ \Psi]\ a \Rightarrow^r_{\rho,\kappa} \text{send } \Phi\ t\text{c}\ m\ c' \text{ fail}}$$

$$\textsc{Comm (Send, Connect)}\quad \frac{t \notin \kappa \qquad a \Rightarrow^r_{\rho,\kappa\cup\{t\}} c'}{[r \to t : \Phi\ m\ \Psi]\ a \Rightarrow^r_{\rho,\kappa} \text{let } t\text{c} = \text{connect } (t\text{:name}) \text{ in send } \Phi\ t\text{c}\ m\ c' \text{ fail}}$$

$$\textsc{Comm (Receive, Connected)}\quad \frac{f \in \kappa \qquad a \Rightarrow^r_{\rho,\kappa} c'}{[f \to r : \Phi\ m\ \Psi]\ a \Rightarrow^r_{\rho,\kappa} \text{recv } f\text{c}\ m\ \Psi\ c' \text{ fail}}$$

$$\textsc{Comm (Receive, Connect)}\quad \frac{f \notin \kappa \qquad a \Rightarrow^r_{\rho,\kappa\cup\{t\}} c'}{[f \to r : \Phi\ m\ \Psi]\ a \Rightarrow^r_{\rho,\kappa} \text{let } f\text{c} = \text{accept in recv } f\text{c}\ m\ \Psi\ c' \text{ fail}}$$

$$\textsc{Comm (Skip)}\quad \frac{f \neq r \qquad t \neq r \qquad a \Rightarrow^r_{\rho,\kappa} c}{[f \to t : \Phi\ m\ \Psi]\ a \Rightarrow^r_{\rho,\kappa} c}$$

$$\textsc{New}\quad \frac{a \Rightarrow^r_{\rho,\kappa} c'}{\text{let } v\ @\ r = \text{new } nt\ a \Rightarrow^r_{\rho,\kappa} \text{let } v = \text{new } nt \text{ in } c'}$$

$$\textsc{New (Skip)}\quad \frac{ar \neq r \qquad a \Rightarrow^r_{\rho,\kappa} c}{\text{let } v\ @\ ar = \text{new } nt\ a \Rightarrow^r_{\rho,\kappa} c}$$

$$\textsc{Bind}\quad \frac{a \Rightarrow^r_{\rho,\kappa} c'}{\text{bind } v\ @\ r \text{ to } m\ a \Rightarrow^r_{\rho,\kappa} \text{let } v = m \text{ in } c'}$$

$$\textsc{Bind (Skip)}\quad \frac{ar \neq r \qquad a \Rightarrow^r_{\rho,\kappa} c}{\text{bind } v\ @\ ar \text{ to } m\ a \Rightarrow^r_{\rho,\kappa} c}$$

$$\textsc{Match}\quad \frac{a \Rightarrow^r_{\rho,\kappa} c'}{\text{match } v\ @\ r \text{ with } m\ a \Rightarrow^r_{\rho,\kappa} \text{match } v\ m\ \_\ c' \text{ fail}}$$

$$\textsc{Match (Skip)}\quad \frac{ar \neq r \qquad a \Rightarrow^r_{\rho,\kappa} c}{\text{match } v\ @\ ar \text{ with } m\ a \Rightarrow^r_{\rho,\kappa} c}$$

$$\textsc{Derive}\quad \frac{a \Rightarrow^r_{\rho,\kappa} c'}{\text{derive } \Phi\ @\ r\ a \Rightarrow^r_{\rho,\kappa} \text{derive } \Phi\ c' \text{ fail}}$$

$$\textsc{Derive (Skip)}\quad \frac{ar \neq r \qquad a \Rightarrow^r_{\rho,\kappa} c}{\text{derive } \Phi\ @\ r\ a \Rightarrow^r_{\rho,\kappa} c}$$

Figure 6.8: End-point projection of WPPL into CPPL

```
1 a(a:name, b, s:name, kas) (kab)
2  let na = new nonce in
3  let chanb = connect b in
4  send _ -> chanb (a, na) then
5  let chans = accept in
6  recv chans ({| b, kab, na, nb:nonce |} kas, stob:msg) -> _
7  then let atob = {| nb |} kab in
8  send _ -> chanb (stob, atob) then return
9  else fail else fail else fail end
```

Figure 6.9: Compilation of Yahalom role `A` into CPPL

```
1 (spec ([a (a b s kas) (kab)]
2        [b (b s kbs) (kab)] [s (a b s kas kbs) ()])
3  [a -> s : a, b, na:nonce]
4  [s -> b : {|a, b, na, kab|} kas, {|a, b, na, kab|} kbs]
5  [b -> a : {|a, b, na, kab|} kas, {|na|} kab, nb:nonce]
6  [a -> b : {|nb|} kab] .)
```

Figure 6.10: Kao Chow in WPPL

```
  (spec ([a (a b s kas) (kab)]
         [b (b s kbs) (kab)] [s (a b s kas kbs) (kab)])
3' let na = new nonce
   [a -> s : a, b, na]
4' let kab = new symkey
   [s -> b : {|a, b, na, kab|} kas, {|a, b, na, kab|} kbs]
5' let nb = new nonce
   [b -> a : {|a, b, na, kab|} kas, {|na|} kab, nb]
   [a -> b : {|nb|} kab] .)
```

Figure 6.11: Kao Chow in WPPL after step one

```
   (spec ([a (a b s kas) (kab)]
          [b (b s kbs) (kab)] [s (a b s kas kbs) (kab)])
    let na = new nonce
    [a -> s : a, b, na]
    let kab = new symkey
4'' bind msg0 = {|a, b, na, kab|} kas
4'' bind msg1  = {|a, b, na, kab|} kbs
    [s -> b : msg0, msg1]
    let nb = new nonce
5'' bind msg2 = {|na|} kab
    [b -> a : msg0, msg2, nb]
6'' bind msg3 = {|nb|} kab
    [a -> b : msg3] .)
```

Figure 6.12: Kao Chow in WPPL after step two

```
    (spec ([a (a b s kas) (kab)]
           [b (b s kbs) (kab)] [s (a b s kas kbs) (kab)])
     let na = new nonce
     [a -> s : a, b, na]
     let kab = new symkey
     bind msg0 = {|a, b, na, kab|} kas
     bind msg1 = {|a, b, na, kab|} kbs
     [s -> b : msg0, msg1]
5''' match msg1 with {|a, b, na, kab|} kbs
     let nb = new nonce
     bind msg2 = {|na|} kab
     [b -> a : msg0, msg2, nb]
6''' match msg0 with {|a, b, na, kab|} kas
6''' match msg2 with {|na|} kab
     bind msg3 = {|nb|} kab
     [a -> b : msg3]
7''' match msg3 with {|nb|} kab .)
```

Figure 6.13: Kao Chow in WPPL after step three

Part III

The Analyses

Chapter 7

Dispatching

In this part of our work, we begin to discuss our protocol analyses. These chapters require an understanding of CPPL and strand spaces (Chapters 4 and 3 respectively).

7.1 Problem and Motivation

A fundamental aspect of a cryptographic protocol is the set of messages that it may accept. Protocol specifications contain patterns that specify the shape of many messages they accept. These patterns can be seen to describe an infinite set of messages, because the variables that appear in them may be bound to innumerable values. We call this set a protocol's *message space*.

There is a history of attacks on protocols based on the use of (parts of) messages of one protocol as (parts of) messages of another protocol [97, 58, 81, 10, 98]. These attacks, called type-flaw (or type-confusion) attacks, depend fundamentally on the protocol relation of *message space overlap*. If the message spaces of two protocols overlap, then there is at least one session of each protocol where at least one message could be accepted by both protocols. This property, however, is more general than a "presence of type-flaw attack" property, because not all overlaps are indications of successful attacks. (In fact, it is common for new versions of a protocol to contain many similar messages, where only the messages with problems were changed.)

The message space overlap property does not only give us insight into the protocol and its relation to other protocols; it also provides a test for a fundamental deployment property: dispatchability. We define *dispatchability* as the ability for a message dispatch to *unambiguously* deliver incoming protocol messages to the proper protocol (session.) This basic property is necessary for servers to provide concurrency and support for many protocol clients.

Servers typically rely on TCP for this property. They assign a different TCP port for each protocol and trust the operating system's TCP implementation to do the dispatching. However, as more and more services are deployed as cryptographic protocols atop existing Web protocols (e.g., SOAP), more explicit methods of

distinguishing protocol messages must be used. Furthermore, by leaving this essential information implicit, it is not included in the formally verified portion of the protocol specification. This means that the protocol that is actually used is *not* the one that is verified. Finally, TCP's notion of session may not match the protocol's: this is particularly problematic in protocols with more than two participants that are not simply compositions of two-party protocols.

Notice that message space overlap implies that dispatchability is not achievable. If there is a message, M, that could be accepted by session p and session q of some protocols, then what would a dispatcher do if delivered M? A faulty identification might cause the actual (though unintended) recipient to go into an inconsistent state, or even leak information, while the intended recipient starves. It cannot *unambiguously* deliver the message, and therefore, is not a correct dispatcher. We present a dispatching algorithm that correctly delivers messages if there is no message space overlap. This algorithm provides proof that the lack of message space overlap implies dispatchability.

We present an analysis that determines whether the message spaces of two protocols (sessions of a protocol) overlap. We also present an analysis, phrased as an optimized dispatcher, that determines *why* there is no overlap between two spaces, by finding the largest abstractions of two protocols for which there is no overlap. We present our analysis of protocols from the SPORE protocol repository [114] and show how studying them provides insights to improve our analyses.

Outline. We develop the message space overlap checker in Sec. 7.2. We present our dispatching algorithms in Sec. 7.3, then develop the second analysis that informs our improved dispatcher in Sec. 7.4. Finally, we discuss related work and our conclusions.

7.2 Analysis

In this section, we determine when there is a message that could be accepted by two sessions of two protocols. This general analysis can then be specialized to the case of two sessions of one protocol by comparing a protocol with itself.

The strand space model of protocol is aptly suited for this problem. We can read off, from the strand of a protocol, each message pattern it accepts by looking at the strand descriptions marked $-$. For example, the strand $+m_1, \_ \rightarrow -m_2, \_ \rightarrow -m_3, \_ \rightarrow$. accepts messages with patterns m_2 and m_3. We denote this set of message patterns as $\mathcal{M}(s)$ for strand s.

Each message pattern m describes an infinite set of messages (one for each instantiation of the variables in m) that would be accepted at that point of the protocol. If we could compare the sets of two patterns, then we could easily lift this analysis to two protocols s and s' by checking each pattern in $\mathcal{M}(s)$ against each pattern in $\mathcal{M}(s')$. The essence of our problem is therefore determining when some message pattern m "overlaps" with another message pattern m', i.e., when there is some actual message M that could be matched by both m and m'. We call this analysis match.

7.2.1 Defining match

We have multiple options when defining match. We could assume that the *structure* of message patterns are potentially ambiguous. That is, we could assume that (m_1, m_2) could possibly overlap with $hash(m_3)$ or $\{m_4\}_k$. We will *not* do this. We assume that messages are serialized using unambiguous encoding schemes [72]. Concrete protocol implementations that do not conform to this assumption may have type-flaw attacks [97, 58, 81, 10, 98].[1]

This initial consideration shrinks the design space of match: message patterns must have identical structure for them to possibly overlap. There are two important caveats: variables with type *msg* and bind patterns ($< v = m >$). In the first case, we treat such variables as "wildcards", because they will accept any message when used a pattern. In the second case, we ignore the variable binding and use the sub-pattern m in the comparison.

Given this structural means of determining when two message patterns potentially overlap, all that remains is to specify when to consider two variables as potentially overlapping. The simplest strategy is to assume that if the types of two variables are the same, then it is possible that each could refer to the same value. We call this strategy type-based and write it match_τ. Its definition is given in Figure 7.1.

Correctness. match_τ is correct if it soundly approximates message space overlap, i.e., if $\neg\mathsf{match}_\tau mm'$ then there is no overlap between the possible messages accepted by pattern m and pattern m'. This implies that $\mathsf{match}_\tau mm'$ should not be read as "Every message accepted by m is accepted by m'" (or vice versa), because there are some environments (and therefore protocol sessions) where there can be no overlap between messages. For example, the pattern x does not overlap with y if x is bound to 2 and y is bound to 3. But, there is at least one environment pair where there is at least one message that *is* accepted by both: when x and y are bound to 2 and the message is 2.

We do not prove that match_τ is complete, i.e., if there is a message space overlap then $\mathsf{match}_\tau pm$. (In fact, it is not, as the further development below demonstrates.)

7.2.2 Evaluating match_τ

The theorem prover can tell us if match_τ is well-defined, decidable, and if it has other desirable properties. But, it cannot tell us if the analysis is useful. We address the utility of the analysis by running it on a large number of pairs of protocol roles.

We have encoded 121 protocol roles from forty-three protocol definitions found in the Security Protocols Open Repository [114] (SPORE) in CPPL. For each role, our analysis generates every possible strand interpretation of the role, then compares each message pattern with those of another role. We find that, when using match_τ, 15.7% of protocol role pairs are non-overlapping; i.e., for 84.3% of the pairs, there is some message that is accepted by both roles in some run. This seems like an extravagantly highly number.

[1] The implementations we produce *do* conform to the assumption.

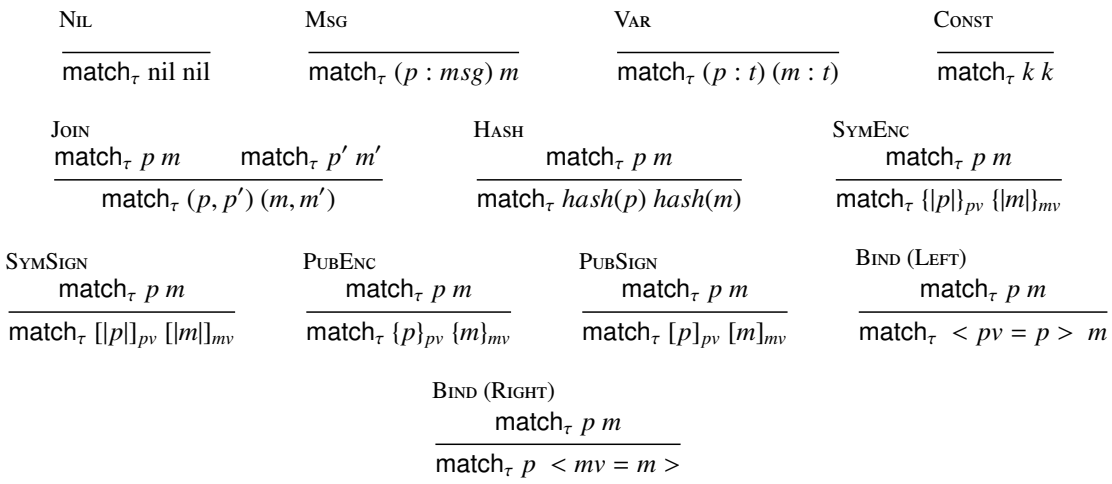


Figure 7.1: Definition of match_τ

If we actually look at the source of many protocols in CPPL, we learn why there are such poor results with match_τ. It turns out that many protocols have the following form:

```
1 recv chan (m1, m:msg) -> _ then
  ...
n match m m2 then
```

where m_1 and m_2 are particular patterns, such as (price, p) or $\{m_1\}_k$.

Consider how match_τ would compare this message with another: because it contains a wildcard message (with type msg), it is possible for *any* message to be accepted. This tells us that the specificity of the protocol role deeply impacts the efficacy of our analysis. In the next section, we develop a transformation on protocol roles that increases their specificity. We will find that it greatly improves the performance of match_τ.

7.2.3 Message Specificity

Suppose we have a protocol the following protocol role:

```
1  recv ch (m1, a) -> _
2  then let nc = new nonce in
3  match m1 {|b, k'|} k -> _
```

If this protocol were slightly different, then we could execute it with more partners. In particular, if it were:

```
1' recv ch (<m1={|b, k'|} k>, a) -> _
```

$$\textsc{Nil}\quad \mathsf{subst}\ x\ xm\ \mathrm{nil} = \mathrm{nil} \qquad \textsc{Var}\quad \mathsf{subst}\ x\ xm\ x =< x = xm > \qquad \textsc{Var (Skip)}\quad \frac{x \neq v}{\mathsf{subst}\ x\ xm\ v = v} \qquad \textsc{Const}\quad \mathsf{subst}\ x\ xm\ k = k$$

$$\textsc{Join}\quad \frac{\mathsf{subst}\ x\ xm\ m_1 = m_1' \qquad \mathsf{subst}\ x\ xm\ m_2 = m_2'}{\mathsf{subst}\ x\ xm\ (m_1, m_2) = (m_1', m_2')} \qquad \textsc{Hash}\quad \mathsf{subst}\ x\ xm\ hash(m) = hash(m)$$

$$\textsc{SymEnc}\quad \frac{\mathsf{subst}\ x\ xm\ m = m'}{\mathsf{subst}\ x\ xm\ \{|m|\}_v = \{|m'|\}_v} \qquad \textsc{SymSign}\quad \frac{\mathsf{subst}\ x\ xm\ m = m'}{\mathsf{subst}\ x\ xm\ [|m|]_v = [|m'|]_v} \qquad \textsc{PubEnc}\quad \frac{\mathsf{subst}\ x\ xm\ m = m'}{\mathsf{subst}\ x\ xm\ \{m\}_v = \{m'\}_v}$$

$$\textsc{PubSign}\quad \frac{\mathsf{subst}\ x\ xm\ m = m'}{\mathsf{subst}\ x\ xm\ [m]_v = [m']_v} \qquad \textsc{Bind}\quad \frac{\mathsf{subst}\ x\ xm\ m = m'}{\mathsf{subst}\ x\ xm\ \ < v = m >=< v = m' >}$$

Figure 7.2: subst Definition (Messages)

```
2  then let nc = new nonce in
3  match m1 {|b, k'|} k -> _
```

In this modified protocol, the wildcard message `m1` on line 1 is replaced by a *more specific* pattern on line 1′. We say that message pattern m_1 is more specific than message pattern m_2, if for all message patterns M, $\mathsf{match}_\tau\ m_1\ M$ implies $\mathsf{match}_\tau\ m_2\ M$, i.e., every message that is accepted by m_1 is accepted by m_2. (Notice that this is a reflexive and transitive relation.)

Our transformation, called foldm, works as follows: for each message reception point where message m is received, record the environment before reception as σ_m, then inspect the rest of the role for pattern points where identifier i is compared with pattern p such that $\sigma_n \vdash_r p$, and replace each occurrence of i in m with $< i = p >$. Refer to the definition in Figures 7.3, 7.2, and 7.4.

We prove the following theorems about this transformation:

Theorem 12. *(`foldm_proc_type`) If* $\vdash$ *proc then* $\vdash$ foldm *proc.*

Proof summary. This is proved by simple induction of the structure of program. The pivotal step is the receive rule where the pattern is changed. This is safe because the pattern matching is only folded back if it is well-formed in the same environment. (For example, this would not be the case if in our example the pattern matching used *nc*.) □

Theorem 13. *(`foldm_proc_more_specific`) For every pattern in proc, there is a corresponding, more specific pattern in* foldm *proc.*

Proof summary. This proof has similar structure to Theorem 12. The conclusion is justified in the pivotal step because i is always replaced with $< i = p >$, which is more specific than i, for any p. □

$$\textsc{Fail}\quad \mathsf{subst}\ \sigma\ sm\ \mathrm{fail} = sm$$

$$\textsc{Return}\quad \mathsf{subst}\ \sigma\ sm\ (\mathrm{return}\ \Phi\ v^*) = sm$$

$$\textsc{Let}\quad \frac{\mathsf{subst}\ \sigma\ sm\ c = sm'}{\mathsf{foldm}\ \sigma\ sm\ (\mathrm{let}\ v = lv\ \mathrm{in}\ c) = sm'}$$

$$\textsc{Derive}\quad \frac{\mathsf{subst}\ \sigma\ sm\ sk = sm' \qquad \mathsf{subst}\ \sigma\ sm'\ fk = sm''}{\mathsf{subst}\ \sigma\ sm\ (\mathrm{derive}\ \Phi\ sk\ fk) = sm''}$$

$$\textsc{Send}\quad \frac{\mathsf{subst}\ \sigma\ sm\ sk = sm' \qquad \mathsf{subst}\ \sigma\ sm'\ fk = sm''}{\mathsf{subst}\ \sigma\ sm\ (\mathrm{send}\ \Phi\ v\ m\ sk\ fk) = sm''}$$

$$\textsc{Receive}\quad \frac{\mathsf{subst}\ \sigma\ sm\ sk = sm' \qquad \mathsf{subst}\ \sigma\ sm'\ fk = sm''}{\mathsf{subst}\ \sigma\ sm\ (\mathrm{recv}\ v\ m\ \Psi\ sk\ fk) = sm''}$$

$$\textsc{Call}\quad \frac{\mathsf{subst}\ \sigma\ sm\ sk = sm' \qquad \mathsf{subst}\ \sigma\ sm'\ fk = sm''}{\mathsf{subst}\ \sigma\ sm\ (\mathrm{call}\ \Phi\ x\ v^*\ u^*\ \Psi\ sk\ fk) = sm''}$$

$$\textsc{Match (Well)}\quad \frac{\sigma \vdash_r m \qquad \mathsf{subst}\ v\ m\ sm = sm' \qquad \mathsf{subst}\ \sigma\ sm'\ sk = sm'' \qquad \mathsf{subst}\ \sigma\ sm''\ fk = sm'''}{\mathsf{subst}\ \sigma\ sm\ (\mathrm{match}\ v\ m\ \Psi\ sk\ fk) = sm'''}$$

$$\textsc{Match (Ill)}\quad \frac{\sigma \nvdash_r m \qquad \mathsf{subst}\ \sigma\ sm\ sk = sm' \qquad \mathsf{subst}\ \sigma\ sm'\ fk = sm''}{\mathsf{subst}\ \sigma\ sm\ (\mathrm{match}\ v\ m\ \Psi\ sk\ fk) = sm''}$$

Figure 7.3: subst Definition

Preservation. We must also ensure that this transformation preserves the semantics of the protocol in a meaningful way. However, since we are clearly changing the set of messages accepted by the protocol (requiring them to be more specific), the transformed protocol does not have the same meaning.

The fundamental issue is: how is the protocol meaning different? Recall that the meaning of a protocol is a set of strands that represent potential runs. This is smaller after the transformation. However, if we only consider the runs that end in success—those runs where whenever a message matching pattern p is expected such a message is provided—then there is no difference in protocol behavior.

Why? Consider the example from above. Suppose that a message M matches the pattern `(m1, a)` is provided at step 1 in the original protocol and the rest of protocol executes successfully. Then, m_1 *must* match the pattern `{|b, k'|} k`. Therefore, the message M must match the pattern `(<m1={|b, k'|} k>, a)`. Therefore, if the same message was sent to the transformed protocol, the protocol would execute successfully. This always holds because the transformation *always* results in more specific patterns that have exactly this property.

What happens to runs that fail in the untransformed protocol? They continue to fail in the transformed protocol, but may fail *differently*. Suppose that a message M is delivered to the example protocol at step 1 and the protocol fails. It either fails at step 1 or step 3. If it fails at step 1, then it does not match the pattern `(m1,a)`, and also does not match the pattern `(<m1={|b, k'|} k>, a)`. Therefore it fails at step 1 in the transformed protocol as well. If it fails at step 3, then the left component of the message M does not match the pattern `{|b,k'|} k`. Therefore, the transformed protocol will fail at step 1 for the very same reason.

$$\textsc{Fail}\quad \mathsf{foldm}\ \sigma\ \text{fail} = \text{fail} \qquad \textsc{Return}\quad \mathsf{foldm}\ \sigma\ (\text{return}\ \Phi\ v^*) = \text{return}\ \Phi\ v^* \qquad \textsc{Let}\quad \frac{\mathsf{foldm}\ (\sigma \cup \{v\})\ c = c'}{\mathsf{foldm}\ \sigma\ (\text{let}\ v = lv\ \text{in}\ c) = \text{let}\ v = lv\ \text{in}\ c'}$$

$$\textsc{Derive}\quad \frac{\mathsf{foldm}\ (\sigma \cup bound(\Phi))\ sk = sk' \qquad \mathsf{foldm}\ \sigma\ fk = fk'}{\mathsf{foldm}\ \sigma\ (\text{derive}\ \Phi\ sk\ fk) = \text{derive}\ \Phi\ sk'\ fk'}$$

$$\textsc{Send}\quad \frac{\mathsf{foldm}\ (\sigma \cup bound(\Phi))\ sk = sk' \qquad \mathsf{foldm}\ \sigma\ fk = fk'}{\mathsf{foldm}\ \sigma\ (\text{send}\ \Phi\ v\ m\ sk\ fk) = \text{send}\ \Phi\ v\ m\ sk'\ fk'}$$

$$\textsc{Receive}\quad \frac{\mathsf{foldm}\ (\sigma \cup bound(m) \cup bound(\Psi))\ sk = sk' \qquad \mathsf{subst}\ \sigma\ m = m' \qquad \mathsf{foldm}\ \sigma\ fk = fk'}{\mathsf{foldm}\ \sigma\ (\text{recv}\ v\ m\ \Psi\ sk\ fk) = \text{recv}\ v\ m'\ \Psi\ sk'\ fk'}$$

$$\textsc{Call}\quad \frac{\mathsf{foldm}\ (\sigma \cup bound(\Phi) \cup u^* \cup bound(\Psi))\ sk = sk' \qquad \mathsf{foldm}\ \sigma\ fk = fk'}{\mathsf{foldm}\ \sigma\ (\text{call}\ \Phi\ x\ v^*\ u^*\ \Psi\ sk\ fk) = \text{call}\ \Phi\ x\ v^*\ u^*\ \Psi\ sk'\ fk'}$$

$$\textsc{Match}\quad \frac{\mathsf{foldm}\ (\sigma \cup bound(m) \cup bound(\Psi))\ sk = sk' \qquad \mathsf{foldm}\ \sigma\ fk = fk'}{\mathsf{foldm}\ \sigma\ (\text{match}\ v\ m\ \Psi\ sk\ fk) = \text{match}\ v\ m\ \Psi\ sk'\ fk'}$$

$$\textsc{Proc}\quad \frac{\mathsf{foldm}\ (v^* \cup bound(\Psi))\ c = c'}{\mathsf{foldm}\ (\text{proc}\ x\ v^*\ \Psi\ c) = \text{proc}\ x\ v^*\ \Psi\ c'}$$

Figure 7.4: foldm Definition

In general, the transformed protocol's behavior is identical, *modulo failure*. If the same sequence of external messages is delivered to a transformed role, then it will either (a) succeed, like the untransformed counter-part; or (b) fail earlier, because some failing pattern matching was moved earlier in the protocol. Semantically, this means that the set of strand bundles are protocol can be a part of is smaller.

It is crucial to actually execute the transformed protocol. If the unmodified protocol is used, then it is certainly possible for the wrong recipient to receive a message and then fail when the more specific pattern matching is attempted.

Adversary. This transformation decreases the amount of harm the adversary can do, or does not change it. Since the only difference in behavior is that faulty messages are noticed sooner, whatever action the principal would have taken before performing the lifted pattern matching is not done. Therefore, the principal does *less* before failing, and therefore the "hooks" for the adversary are *decreased*. Of course, for any particular protocol, these hooks may or may not be useful, but in general, there are fewer hooks.

7.2.4 Evaluating foldm

When we apply foldm to our test suite of 121 protocol roles, then run the match_τ analysis, we find that the percentage of non-overlapping role pairs increases from 15.7% to 61%.

This means that for 61% of pairs of protocol roles from our repository, it is always possible to unambiguously deliver a message to a single protocol handler. However, when we just look at the special case of comparing a role with itself, i.e., determining if it is possible to dispatch to session correctly, we find that 0% of the roles have this property according to match_τ.

This is an unsurprising result. Every message pattern p is exactly the same as itself. Therefore, match_τ will resolve that p has the same shape as p, and could potentially accept the same messages.

The problem is that match_τ only looks at the two patterns. It does not consider the context in which they appear: a cryptographic protocol that may make special assumptions about the values bound to certain variables. In particular, some values are assumed to be unique. For example, in many protocols *nonces* (*n*umbers used *once*) are generated randomly and used to prevent replay attacks and conduct authentication tests [47]. In the next section, we describe how to incorporate assumptions about uniqueness in our analysis.

7.2.5 Relying on Uniqueness

In the Andrew Secure RPC role (Fig. 4.2), the message received on line 6 must match the pattern $\{|nb|\}_{kab}$, where *nb* is a nonce that was freshly generated on line 4. This means that *no two sessions* of this role could accept the same message at line 6, because each is waiting for a *different unique* value for *nb*.

We call the version of our analysis that incorporates information about uniqueness match_δ. Whenever it is comparing a variable u from protocol α and a variable v from protocol β, if u is in the set of unique values generated by α or v is in the set of unique values generated by β, then the two are assumed not to match, regardless of anything else about the variables. In all other cases, two variables are assumed to be potentially overlapping, i.e., the types are ignored, unlike match_τ.

7.2.6 Evaluating match_δ

When we apply match_δ to our test suite, we find that the percentage of non-overlapping sessions is 0.8%. After applying the foldm transformation, this increases to 14.8%.

If we look at the other 85.2% of the protocols, is there anything more that can be incorporated into the analysis? There is. The first action of many protocol roles is to receive some particular initiation message. Since this is the *first* thing the role does, it cannot possibly contain a value uniquely generated by the role. Therefore, the match_δ analysis will not be able to find a unique value that distinguishes the session the message is meant for. In the next section, we will discuss how to get around this difficulty.

$$\textsc{Nil}\ \frac{}{\mathsf{match}_\delta\ \alpha\ \beta\ \mathrm{nil}\ \mathrm{nil}} \qquad \textsc{Msg}\ \frac{}{\mathsf{match}_\delta\ \alpha\ \beta\ (p : msg)\ m} \qquad \textsc{Var}\ \frac{p \notin \alpha \qquad m \notin \beta}{\mathsf{match}_\delta\ \alpha\ \beta\ (p : t)\ (m : t)} \qquad \textsc{Const}\ \frac{}{\mathsf{match}_\delta\ \alpha\ \beta\ k\ k}$$

$$\textsc{Join}\ \frac{\mathsf{match}_\delta\ \alpha\ \beta\ p\ m \qquad \mathsf{match}_\delta\ \alpha\ \beta\ p'\ m'}{\mathsf{match}_\delta\ \alpha\ \beta\ (p, p')\ (m, m')} \qquad \textsc{Hash}\ \frac{\mathsf{match}_\delta\ \alpha\ \beta\ p\ m}{\mathsf{match}_\delta\ \alpha\ \beta\ hash(p)\ hash(m)}$$

$$\textsc{SymEnc}\ \frac{pv \notin \alpha \qquad mv \notin \beta \qquad \mathsf{match}_\delta\ \alpha\ \beta\ p\ m}{\mathsf{match}_\delta\ \alpha\ \beta\ \{|p|\}_{pv}\ \{|m|\}_{mv}} \qquad \textsc{SymSign}\ \frac{pv \notin \alpha \qquad mv \notin \beta \qquad \mathsf{match}_\delta\ \alpha\ \beta\ p\ m}{\mathsf{match}_\delta\ \alpha\ \beta\ [|p|]_{pv}\ [|m|]_{mv}}$$

$$\textsc{PubEnc}\ \frac{pv \notin \alpha \qquad mv \notin \beta \qquad \mathsf{match}_\delta\ \alpha\ \beta\ p\ m}{\mathsf{match}_\delta\ \alpha\ \beta\ \{p\}_{pv}\ \{m\}_{mv}} \qquad \textsc{PubSign}\ \frac{pv \notin \alpha \qquad mv \notin \beta \qquad \mathsf{match}_\delta\ \alpha\ \beta\ p\ m}{\mathsf{match}_\delta\ \alpha\ \beta\ [p]_{pv}\ [m]_{mv}}$$

$$\textsc{Bind (Left)}\ \frac{\mathsf{match}_\delta\ \alpha\ \beta\ p\ m}{\mathsf{match}_\delta\ \alpha\ \beta\ <pv = p>\ m} \qquad \textsc{Bind (Right)}\ \frac{\mathsf{match}_\delta\ \alpha\ \beta\ p\ m}{\mathsf{match}_\delta\ \alpha\ \beta\ p\ <mv = m>}$$

Figure 7.5: Definition of match_δ

7.2.7 Handling Initial Messages

The first thing the Andrew Secure RPC role (Fig. 4.2) does, on line 3, is to receive a certain message: $(a, \{|na|\}_{kab})$. Since this message does not contain any value uniquely generated for the active session role, this means that the initial message of one session of Andrew Secure RPC can be confused with the initial message of another session of Andrew Secure RPC. Actually, this is contradictory: the initial message is what *creates* sessions, so by definition it cannot be confused with itself across sessions.

Therefore, we can safely ignore the first message of a protocol role, if it is not preceded by any other action, for the purposes of determining the dispatchability of a protocol role's sessions. We must, of course, compare the initial message with all *other* messages, to ensure that the initial message cannot be confused with, e.g., the third message, but we do not need to compare it with itself. When we use this insight with the match_δ analysis, we write it as $\mathsf{match}_{\iota(\delta)}$.

7.2.8 Evaluating $\mathsf{match}_{\iota(\delta)}$

When we apply $\mathsf{match}_{\iota(\delta)}$ to our test suite, with the foldm transformation, the percentage non-overlapping sessions is 62.8%.

Table 7.1a presents the results when analyzing each pair of protocol roles. Interestingly, unique values are *not* very useful when comparing roles, although they do increase the coverage slightly.

We have inspected the protocols not handled by $\mathsf{match}_{\tau+\delta}$ to determine why the protocol pairs potentially

	match_{τ}	match_{δ}	$\mathsf{match}_{\tau+\delta}$
initial	15.7%	10.0%	15.8%
foldm	61.0%	55.2%	62.1%

(a) Non-overlapping Protocol Role Pairs

	match_{τ}	match_{δ}	$\mathsf{match}_{\tau+\delta}$
initial	0.0%	00.8%	00.8%
foldm	0.0%	14.8%	14.8%
ι + foldm	31.4%	62.8%	62.8%

(b) Non-overlapping Protocol Role Sessions

Table 7.1: Analysis Results

accept the same message.

1. Protocols with similar goals and similar techniques for achieving those goals typically have the same initial message. Examples include the Neumann Stubblebine [108, 65], Kao Chow [69], and Yahalom [15, 88, 113] protocol families.

2. Different versions of the same protocol will often have very similar messages, typically in the initial message, though not always—often these protocols are modified by making tiny changes, so the other messages remain identical. A good example is the Yahalom [15, 88, 113] family of protocols.

3. Some protocols have messages that cannot be refined by foldm, because the key necessary to decrypt some message component must be received from another message or from a trust management database query. This leaves a message component that will match any other message, so such protocols cannot be paired with a large number of other protocols. One example is the S role of some Yahalom [15, 88, 113] variants.

4. For many protocols, there is dependence among the pattern-matching in the continuation of message reception. (One example is the P role of the Woo Lam Mutual [143] protocol.) As a result of this, only the independent pattern is substituted into the original message reception pattern. This leaves a variable in the pattern that matches all messages. We could remove this problem with an unsound version of the protocol refinement transformation, foldm, that used some related notion of well-formed-ness that allowed multiple passes over the input pattern in pattern-matching. However, we believe soundness is an important property and that our current transformation is good enough.

Table 7.1b presents the results when analyzing the sessions of each protocol role. It may seem odd that the $\mathsf{match}_{\iota(\tau)}$ analysis is able to verify any sessions, given our argument against match_{τ}. Why should removing the initial message make any difference? In 31.4% of the protocols, the protocol *only* receive a single, initial message. We have also inspected the protocols that most permissive session-based analysis rules out.

1. Some messages simply do not contain a unique value. A prominent example is the A role of many variants of the Andrew Secure RPC [122, 15, 84] protocol.

2. Some roles have the same problems listed above as (3) and (4), except that in these instances the lack of further refinement hides a unique value. One example is the C role of the Splice/AS [144] protocol.

7.3 Dispatching

Our analysis determines for a pair of protocols (recall that sessions are a special case) when there is no message that could be confused during any run of the two protocols. We can use this property to build a dispatching algorithm. The algorithm is very simple: Forward every incoming message to every protocol handler. (In the case of sessions, we must specially recognize initial messages and create a new session; otherwise, we forward the messages to each session.)

This algorithm is correct, because every message that is accepted by *some* protocol (session) is only accepted by *one* protocol (session), according to the property our analysis determines.

This algorithm makes no attempt to determine which protocol an incoming message is actually intended for. This is a source of inefficiency.

On a single machine, where "forwarding a message" corresponds to invoking a handling routine, there are two major costs: (1) a linear search through the various protocol/session handlers; and, (2) the CPU cost associated with each of these handlers. In some scenarios, cost 2 is negligible because most network servers are not CPU-bound. However, since we are dealing with *cryptographic* protocols, the cost of performing decryption, only to find an incorrect nonce, etc, is likely to be prohibitive.

In a network load-balancing setting, where "forwarding a message" actually corresponds to using network bandwidth, this algorithm essentially destroys the intended benefits of such a setup.

A better algorithm would be able to keep an index of protocol sessions waiting for messages and compare incoming messages against the patterns on behalf of the underlying sessions. One efficient indexing strategy is a generalized trie [61] on the type of message patterns.

The main problem with the improved algorithm is that it requires *trust* in the dispatcher: the dispatcher must look inside encrypted components of messages to determine which protocol (session) they belong to. In the next section we discuss how to (a) minimize and (b) characterize the amount of trust that must be given to a load-balancer of this sort to perform correct dispatching.

7.4 Optimization

Our task in this section is to determine how much trust, in the form of secret data (e.g., keys), must be given to a load-balancer to inspect incoming messages to the point that they can be distinguished. First, we will formalize how deep a load-balancer can inspect any particular message with a certain amount of information. Second, we describe the optimization process that determines the optimal trust for any pair of protocols. Finally, we formalize the security repercussions of this trust. The end result of this section is a metric of how efficient dispatching can be for a protocol: all protocols should aspire to require no trust in the dispatcher.

$$\textsc{Nil}\quad \text{nil}\downarrow^{\sigma} = \text{nil} \qquad \textsc{Var}\quad v\downarrow^{\sigma} = v \qquad \textsc{Const}\quad k\downarrow^{\sigma} = k \qquad \textsc{Join}\quad (m, m')\downarrow^{\sigma} = (m\downarrow^{\sigma}, m'\downarrow^{\sigma}) \qquad \textsc{Hash}\ \frac{\sigma \vdash_s hash(m)}{hash(m)\downarrow^{\sigma} = hash(m)}$$

$$\textsc{Hash (Wild)}\ \frac{\sigma \nvdash_s hash(m)}{hash(m)\downarrow^{\sigma} = *} \qquad \textsc{SymEnc}\ \frac{k \in \sigma}{\{|m|\}_k\downarrow^{\sigma} = \{|m\downarrow^{\sigma}|\}_k} \qquad \textsc{SymEnc (Wild)}\ \frac{k \notin \sigma}{\{|m|\}_k\downarrow^{\sigma} = *} \qquad \ldots \qquad \textsc{Bind}\quad <v = m>\downarrow^{\sigma} = <v = m\downarrow^{\sigma}>$$

Figure 7.6: Message Redaction

7.4.1 Message Redaction

Suppose that a message is described by the pattern $(a, \{|b|\}_k)$. If the inspector of this message does not know key k, then in general[2], this message is not distinguishable from $(a, *)$. We call this the *redaction* of pattern $(a, \{|b|\}_k)$ under an environment that does not contain k.

We write $m\downarrow^{\sigma}$ as the redaction of message m under σ. This is defined in Figure 7.6. We verify that this operation meets our intuitions through a number of theorems.

Theorem 14. *(`msg_redact_recv_type_recv`) A receiver in environment σ can interpret $m\downarrow^{\sigma}$: for all σ and m, $\sigma \vdash_r m\downarrow^{\sigma}$.*

Proof summary. The definition of redaction follows the rules for well-formedness directly, so this proof is very simple induction on the structure of m. □

Theorem 15. *(`msg_redact_recv_specific`) m is more specific than $m\downarrow^{\sigma}$, i.e., every message that is matched by m is matched by $m\downarrow^{\sigma}$. (Sec. 7.2.3)*

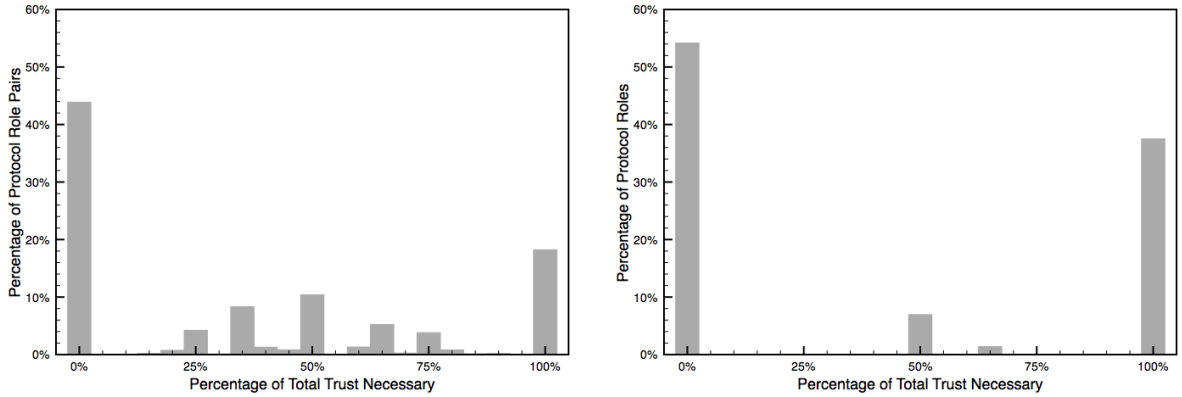
Proof summary. This proved by induction on m and the trivial lemma that any m is more specific than $*$. □

Theorem 16. *(`msg_redact_recv_identity`) $\sigma \vdash_r m$ implies $m\downarrow^{\sigma} = m$.*

Proof summary. This proof is a very simple induction, because the definition of redaction is so close to well-formedness structurally. □

These theorems establish that $m\downarrow^{\sigma}$ captures the view that a load-balancer, that is only trusted with σ, has of a message m. The next task is to minimize σ while ensuring that match can rule out potential message confusion.

[2]There are kinds of encryption that allow parties without knowledge of a key to know that some message is encrypted by *that* key, but still not know the contents of the message.



(a) (b)

Figure 7.7: Trust Optimization Graphs

7.4.2 Minimizing σ

Suppose we are comparing $m = (\{|b|\}_k, \{|c|\}_j)$ with $m' = (\{|b'|\}_{k'}, \{|c'|\}_{j'})$, where b and b' are unique values of their respective protocols, with $\mathsf{match}_{\tau+\delta}$. Because b and b' are unique, the analysis, and therefore the load-balancer, only needs to look at b and b' to ensure that these message patterns cannot describe the same messages. This means that even though the patterns mentions the keys k and j (k' and j'), only k (k') is necessary to distinguish the messages. Another way of putting this is that $m \downarrow^{\{k\}} = (\{|b|\}_k, *)$ does not overlap with $m' \downarrow^{\{k'\}} = (\{|b'|\}_{k'}, *)$ according to $\mathsf{match}_{\tau+\delta}$.

We prove that if m and m' cannot be confused according to match, then there is a computable smallest set σ, such that $m \downarrow^{\sigma}$ also cannot be confused with $m' \downarrow^{\sigma}$ according to match. We prove this by first showing that for all m, there is a set $\mathcal{V}_m$ such that for all σ, $m \downarrow^{\mathcal{V}_m \cup \sigma} = m \downarrow^{\mathcal{V}_m}$.

In other words, there is a "strongest" set for $\downarrow$ that cannot be improved. This set is, of course, the set of σ such that $\sigma \vdash_r m$. Our construction algorithm then considers each subset of $\mathcal{V}_m$ ($\mathcal{V}_{m'}$) and selects the smallest such that the two messages are still distinct after $\downarrow$.

We have run this optimization on our test-suite of 121 protocol roles. Figure 7.7a shows the percentage of pairs of protocols that only require each percentage of their keys. This graph shows that 43% of pairs of protocols do not require *any* trust to properly dispatch. On the other end of the graph, it shows that only 18% of all protocol pairs require complete trust in the load-balancer. Figure 7.7b shows the same statistics for protocol sessions. In this situation, 54% of the protocol roles do not require any trust for the load-balancer to distinguish sessions, while 37% require complete trust. These results were calculated in 7 : 36 minutes (31 milliseconds per pair) and 1.6 seconds (13 milliseconds per role) respectively.

These experiments inform us that it is very fruitful to pursue optimizing the amount of trust given to a load-balancer. However, we have not yet characterized the security considerations of this trust. We do so in the next section.

7.4.3 Managing Trust

In previous sections, we have discussed how much trust to give to a load-balancer so it can dispatch messages correctly. In this section, we provide a mechanism for determining the security impact of that trust.

Recall that a protocol is specified as a *strand*: a list of messages to send and receive. We have formalized "trust" as a set of keys (and other data) to be shared with a load-balancer. We define a strand transformation $\uparrow^k$ that transforms a strand s such that it shares k, by sending a particular message containing k, as soon as possible. (It is trivial to lift $\uparrow^k$ to share multiple values.)

We define $s \uparrow^k$ as follows:

$$(sd \rightarrow s) \uparrow^k = sd \rightarrow +(\mathsf{LB}, v), \_ \rightarrow s \text{ if } k \in bound(sd)$$
$$(sd \rightarrow s) \uparrow^k = sd \rightarrow (s \uparrow^k) \text{ if } k \notin bound(sd)$$
$$. \uparrow^k = .$$

(This definition clearly preserves well-formedness and performs its task.) In this definition LB is a tag that indicates this value is shared with the load-balancer by some means. Depending on the constraints of the environment, this means can be assumed to be perfectly secure or have some specific implementation; e.g., by using a long-term shared key or public-key encryption.

Since $s \uparrow^k$ is a proper strand, it can be analyzed using existing tools and techniques [39, 126, 51, 34]. In particular, the impact of an adversary load-balancer can be easily analyzed.

7.5 Insights

The development of message space overlap analysis and the corollary trust optimization process give us insight about *why* and *how* cryptographic protocol message spaces do not overlap.

The effectiveness of match_τ for pairs of protocols demonstrates that it is primarily *shape* that leads to lack of overlap between different protocols. This corresponds with our intuitions, because protocols typically use dissimilar formats, even if they use similar principles.

The disparity between match_τ and match_δ demonstrates that for pairs of protocol sessions, it is uniquely originating values, like nonces, that provide this property. Again, this corresponds with our intuitions, because nonces are consciously designed to prevent replay attacks and ensure freshness, which corresponds to the goal of identifying sessions.

The statistical differences between these two analyses in different settings allow us to make these conclusions in a coarse way. But, the results of the trust optimization process provide the real answer to the question, "Why do two messages space not overlap?"

When the trust optimization process redacts a message, it is removing the parts of the message that are *not* useful for distinguishing that protocol (session.) This means that what remains *is*, and therefore, the fully

redacted message is *only* what is necessary to ensure there is no message space overlap. Thus, for any two protocols (sessions), it is the trust optimization that gives us insight into why there is no overlap.

7.6 Previous Work

In our past work [94], we specifically addressed the question of when a protocol role supports the use of multiple sessions, but our approach was significantly different. First, in that study we presented a program transformation similar to foldm; however, we did not formalize the correctness of the transformation. Second, we only used the naïve dispatching algorithm and did not investigate a more useful algorithm. Third, we did not consider pairs of protocols. Therefore, the current presentation is much more rigorous, more practical and more general.

Furthermore, our previous problem statement was to inspect protocol role message patterns for the presence of distinguishing (unique) values only. This is clearly incorrect in the case of protocol role pairs. Consider the roles A and B where A accepts the message M_a then $(N_a, *)$ and B accepts the message M_b then $(*, N_b)$, where N_x is a locally generated nonce for x. Each message pattern of each role contains a distinguishing value, so it passes the analysis, but it is not deployable with the other protocol, because it is not possible to unambiguously deliver the message (N_a, N_b) after the message M_a and M_b have been delivered.

It is actually much worse than this: we can encode these two protocols as one protocol: Accept either M_a or M_b, then depending on the first message, accept $(N_a, *)$ or $(*, N_b)$. Our earlier analysis would ignore the initial messages (which is problematic in itself if M_a and M_b overlap), then check all the patterns in each branch, and report success. This is clearly erroneous because it is possible to confuse an "A"-session with a "B"-session.

Our current formalization avoids these problems by directing phrasing the problem in terms of deciding message overlap—the real property of interest, rather than a proxy to it, as distinguishing values were. It is useful to point out, however, that the earlier work was sound for protocol roles that did not contain branching, which is an incredibly large segment of our test suite.

7.7 Related Work

Dispatching. The Guttman and Thayer [49] notion of protocol independence through disjoint encryption and a related work by Cortier et al. [27] study the conditions under which security properties of cryptographic protocols are preserved under composition with one or more other protocols. This is an incredibly important problem, as it ensures that it is *safe* to compose protocols. A fundamental result of their study shows that different protocols must not encrypt similar patterns by the same keys—a similar conclusion to some of our work. However, our work complements theirs by studying if it is *possible*, from an engineering stand-point, to compose protocols, and in particular, how efficient such a dispatch can be. Ideally both of this problems

must be addressed before deployment.

Our problem is tangentially related to a problem in dispatching messages in object-oriented systems [38]. In these systems it is an error when a particular method is ambiguously provided by two super-classes. However, the solutions to both these problems are not very applicable to the other domain.

The problem of detecting type-flaw attacks [97, 58, 81, 10, 98, 30] is similar to ours. These attacks are based on the inability of a protocol message receiver to unambiguously determine the shape of a message. For example, a nonce may be sent where the receiver expects a key; or, a composite message may be given in place of a key; etc. These attacks are often effective when they force a regular participant into using known values as if they were keys. Detecting when a particular attack is a type-flaw attack, or when components of a regular protocol execution may be used in such, is similar to our problem. These analyses try to determine when sent message components can be confused with what a regular participant expects. However, in these circumstances a peculiar notion of message matching is used that captures the ambiguity in bit patterns. Some analyses use size-based matching where any message of n-bits can be accepted by a pattern expecting n-bits; for example, an n-bit nonce can be considered an n-bit key. Others assume that message structure is discernible, but the leaf-types are not, so a nonce paired with a nonce cannot be interpreted as single nonce, but it may be as a nonce paired with a key. Our analysis is similar in spirit, but differs in both the notion of message overlap and the selection of sent and expected messages: we assume that message shapes can be encoded reliably and all expected messages of one protocol with those of another.

Optimization. The problem of optimizing the amount of trust given to a load-balancer is very similar in spirit to ordering of pattern-matching clauses [78] and ordering rules in a firewall or router [7, 93, 77], which are both similar to the decision tree reduction [118] problem. However, our domain is much simpler than the general domain of these problems and the constants are much smaller ($|\mathcal{V}_m|$ is rarely greater than 3 for most protocols), so we are not afflicted with many of the motivating concerns in those areas. Even so, these problems really only serve as guidelines for the actually optimization process, not the formulation of the solution (i.e., $\downarrow_\sigma$).

7.8 Future Work

In the future we will pursue a complete version of the overlapping analysis, thereby removing the necessity of finding interesting protocols with obscure kinds of non-overlap. We are also interested in dynamic form of non-overlap wherein already running protocols (sessions) can prevent new protocols (sessions) from starting that could contain confusing messages. This kind of analysis would have more information (in the form of realized constants for pattern variables), and thereby increase the applicability of this work to other protocols. Finally, we plan on creating a protocol transformation that modifies a protocol by introducing a plain-text nonce, used only for dispatching, into every received message. This transformation would ensure that no trust is required of the load-balancer. The greatest challenge is to ensure that the transformation maintains

the security of the protocol.

7.9 Conclusion

We have presented an analysis (match) that determines if there is an overlap in the message space of different protocols (or sessions of the same protocol.) We have shown how it is important to look at real protocols in the development of this analysis; in our case, the SPORE repository [114]. By looking at real protocols, we learned that it was necessary to (1) refine protocol specifications (foldm); (2) incorporate cryptographic assumptions about unique values (match_δ); and (3) take special consideration of the initial messages of a protocol ($\mathsf{match}_{\iota(\delta)}$).

We have shown how this analysis, and the message space overlap property, can be used to provide the correctness proof of a dispatching algorithm. We have discussed the performance implications of this algorithm and pointed towards the essential features of a better algorithm. We have developed a formalization ($\downarrow_\sigma$) of the "view" a partially-trusted load-balancer has of messages. We have presented an optimization routine that minimizes the amount of trust necessary for match to succeed on a protocol pair. We have presented the results of this analysis for the SPORE repository. We have also formalized the modifications ($\uparrow^k$) that must be made to a protocol to enable trust of a load-balancer. We have discussed how this optimization characterizes *why* there is no overlap between two message spaces.

This work demonstrates static analyses that provide insight into properties of cryptographic protocols that are not strictly related to security, although have some relation to security properties. Our work demonstrates how these properties are useful in building protocol deployments and implementations. Our analysis is feasible is feasible because of the high-level protocol specification style of CPPL. Our analyses are sound and do not compromise the trustworthiness or soundness of our protocol implementations.

Note. A paper based on an early version of the content of this chapter appeared at the World Wide Web conference in 2007.

Chapter 8

Minimal Backups

In this chapter, we discuss an analysis of cryptographic protocols that reveals their internal structure.

8.1 Problem and Motivation

Each cryptographic protocol has a goal. This goal is conveniently expressed as data and *properties* about that data. For example, a two-party key agreement protocol's goal might be a key (the data) that is shared only between two parties (the property). This property expresses the relationship between three pieces of data: the key and the identities of the two parties.

Each step of a protocol can be seen as advancing towards the ultimate goal. Some steps provide necessary data. Some steps provide *evidence* that a property is true. And other steps serve to provide these two kinds of information to other parties.

The evidence of properties *may not be* explicitly in the protocol. Consider an authentication test [51]: protocol role P receives a nonce that uniquely originated in a message sent by P encrypted with Q's public key. That P received the nonce back enables P to conclude that Q received and acted on the message (provided Q's private key is not compromised). The evidence of this property is not explicitly in the protocol. These kinds of properties are often checked in formal verification models, such as strand spaces [39].

The evidence of other properties *are* explicitly in the protocol, in the form of rely-guarantee *formulas* [53]. To use this technique, each message transmission is annotated with formulas that express properties about the transmitted values, and each message reception is annotated with formulas that express properties about the received values. A formula on a transmission *must be* guaranteed, or checked, by the transmitting party, while formulas on reception are assumed to be true. Such a protocol is sound if, in every execution, for every reception, the assumed formulas are guaranteed earlier in the execution (perhaps by another party).

This process of learning information and converging towards the final goal may be made up of smaller protocols that establish contingent, intermediate goals. For example, a protocol to secretly exchange some

```
  (spec ([a (a b kab) (kabn:symkey)]
         [b (a b kab) (kabn)])
1 [a -> b : a, {|na:nonce|} kab]
2 [b -> a : {|na, nb:nonce|} kab]
3 [a -> b : {|nb|} kab]
4 [b -> a : {|kabn, nbn:nonce|} kab] .)
```

Figure 8.1: Andrew Secure RPC Protocol in WPPL

information may begin with a key agreement protocol to produce a session key. This process of sub-protocol invocation may be *implicit*, i.e., the main protocol embeds a sub-protocol pattern, rather than explicitly calling another protocol.

This structure is revealed when we consider what information (data and explicit evidence) is *necessary* at each step of a protocol run. For example, a key agreement protocol may use a nonce N_k to prevent replay attacks; when this protocol is used as a sub-protocol, after the session key is provided, N_k is no longer necessary, i.e., it never appears in message transmissions, receptions, or formula annotations. Data *and* properties can become unnecessary. In the case of the nonce, properties about its freshness may be unnecessary as well.

Thus far we have only discussed why the necessary information at a step of a protocol is valuable; we do not know if it can be known. It seems plausible that we should be able to know what information is *sufficient* for a protocol to continue, because this is essentially a backup of the protocol run. However, the *necessary* information is effectively a *minimal backup*: the smallest backup that allows the protocol to continue.

Our solution is to first define and compute backups of cryptographic protocol runs, then we will prove that they are minimal by a particular ordering. This solution strategy has the added benefit of giving us an "optimal" fault-tolerance strategy in the form of minimal backups.

Since our backups are minimal, and therefore capture the necessary information at a protocol step, they provide a rich diagnostic to protocol researchers. We show how studying the minimal backup can detect certain kinds of flaws in cryptographic protocols. We discuss how the structure of a protocol is revealed by how the minimal backup changes over time. We evaluate a test-suite of protocols [114] using this diagnostic method and discuss our findings. We find (Sec. 8.6) that almost all protocol roles discard over 60% of their available information at some point, thereby demonstrating their structure.

8.2 Informal Solution

In this section we provide the high-level overview of our solution to the minimal backup problem. Afterwards, we develop the formal framework and then present our findings.

Protocols. Figure 8.1 presents the WPPL encoding of the Andrew Secure RPC [122] protocol. Recall that the semantic interpretation of this specification is essentially the set of possible sequences of messages to be sent or received from the network. These messages are decorated with formulas that express properties about values in the messages. In the Andrew protocol, these formulas would express the freshness of the nonces `na`, `nb`, and `nbn` and the key `kabn`. The evidence of these formulas would be the generation time of the value. Executions may compare this time with the current time at their use to ensure the assumptions about the capacity of the adversary are valid. (For example, an execution may assume that a nonce can be guessed after T seconds and ensure that this interval has not passed.)

Backups. The intuitive picture of a protocol as a list of messages gives rise to a natural understanding of a protocol backup: a backup after a step of the protocol is (a) the data necessary to produce or understand the rest of the messages and (b) the evidence of formulas relevant to this data. We formalize this intuition in Section 8.3.

Consider the Andrew protocol from the perspective of the B role. Before the protocol starts, i.e., before step 1, the only necessary data is the initial knowledge of the role. However, between steps 1 and 2, it is necessary to remember `na`, so it can be sent back to role A. Similarly, between steps 2 and 3, it is necessary to remember `nb`, so that A's response can be checked, and that it was uniquely generated locally, so that its freshness can be checked as well.

Minimal Backups. We must define an ordering on backups before we are able to meaningfully describe a backup that is minimal. Intuitively, a backup a is smaller than a backup b, when the messages are physically smaller and the formulas are stronger, i.e., any formula of b can be deduced from the formulas in a. (A formula could express a property such as "A is even", "A is prime", or "A is 2". The latter property is stronger than the first two, since they can both be derived from it.) For details, see Section 8.3.1.

Computing the Minimal Backup. When we compute the minimal backup, we must ensure that (a) minimality is achieved and (b) that the content is available at the correct point in the protocol run. This part of work is highly dependent on our formalization of protocols, so we must simply refer the reader to Section 8.4.1.

After computing the content of the backup, we must calculate the relevant formulas, i.e., the formulas about the necessary data. The interesting part about this is that the formulas and backup content are defined inductively. We ensure that this set is finite and computable in Section 8.4.2.

Outline. We define the notion of protocol backups and provide an ordering on backups in Sec. 8.3. We show how to provably compute the minimal backup for a protocol at any given point in its execution in Sec. 8.4 using the Andrew protocol role as an example. Finally, we discuss our findings, related work, and our conclusions.

8.3 Definition of a Backup

Intuitively, the backup at point p is a message that can be generated at point p and if received later can be used to completely reconstruct the state at p sufficiently to execute the rest of the protocol run. For example, at step 0 of our example protocol (Fig. 3.2), the backup is only contains the initial knowledge of the roles, because the protocol has not done any work. Before step 9, when `ai` and `kabn` are to be sent, it is clearly necessary to know them. The backup should also contain the evidence of any formulas related to `ai` or `kabn`. In this case, the only formula is `new_symkey([kabn:symkey])` from step 7.

This satisfies the second part of our intuition: if we received `ai`, `kabn`, and evidence of `new_symkey([kabn:symkey])`, then we could complete this strand from step 9. Does this backup meet the first part of the intuition? Can it be generated at step 9? It certainly can: `ai` is bound at step 0, `kabn` is bound at step 7, and `new_symkey([kabn:symkey])` is guaranteed (evidence is provided) at step 7.

With this intuition, we will now formally define a backup.

Definition A pair of a message m and the evidence of the properties associated with the formulas fs is the backup of the well-formed strand $\texttt{sl} \rightarrow \texttt{sr}$ after `sl` if and only if:

1. The message m can be generated and the evidence of the formulas in fs can be provided after `sl`: $\emptyset \vdash \texttt{sl} \rightarrow +m, fs \rightarrow$.
2. `sr` can be executed after receiving message m and the evidence of the formulas fs: $\emptyset \vdash -m, fs \rightarrow \texttt{sr}$
3. fs entails all *relevant* formulas: If x is an identifier in `sr`, m, or fs and f is a formula in `sl` that mentions x, then f must be *entailed* by fs.

The first condition corresponds to the intuition that a backup must be generated at the given point in the protocol. The second condition formalizes the intuition that the rest of the protocol must be executable by consulting the backup. The third and final condition corresponds to the intuition that all of the *relevant* properties must be deducible from the backup, where relevance is defined as mentioning any identifier in the backup (m or fs) or the rest of the protocol (`sr`).

Preservation. It may not be immediately clear how or why this definition of a backup preserves the semantic meaning of protocol. The backup assumes that the strand `sl` has executed (per condition one.) The backup message allows the strand `sr` to be executed (per condition two.) Since the protocol run is entirely described by the run of $\texttt{sl} \rightarrow \texttt{sr}$, we must only explain why this is equivalent to $\texttt{sl} \rightarrow +m, fs \rightarrow -m, fs \rightarrow \texttt{sr}$. Clearly the beginning and the end are equivalent. The actual recording and reading of the backup, however, is not so clear. This is where we must assume that the backup is completely and entirely secure and invisible to the outside world: if it were not, then these actions would be potentially dangerous, and therefore, not equivalent. Since we assume that they *are* safe, they can be ignored for the purposes of the semantics of

the protocol. Therefore, using backups preserves meaning precisely: there are no more or less attacks the adversary is capable of.

It is conceivable to not make this assumption and still maintain security. For example, we could encrypt every backup message with a key known only to the principal before recording it. We do not need to be specific about what mechanism is used for this purpose.

8.3.1 Minimality

Since our goal is to compute the *minimal* backup of a protocol at a certain point, it is necessary to define an ordering on backups. We have defined a backup as a pair of a message m and a set of formulas fs, so we must provide an ordering on these, then combine those orderings to construct one ordering on backups.

Intuitively, a message m is smaller than a message m' when it is physically smaller, i.e., uses less bits. However, backup messages are just sequences of values, not general messages, i.e., they do not contain encryption, hashing, etc. Therefore, we abstract the backup *message* to the set of identifiers that appear in the message. We approximate the physical ordering by the subset relation. For example, `(a)` is smaller than `(a,b)`, which is smaller than `(a,b,c)`.

The meaningful partial order on sets of formulas is in terms of entailment. Φ entails Ψ, when for every $\psi in \Psi$, $\Phi \models \psi$. The formula "A is 2" entails the two formulas "A is even" and "A is prime".

Given these definitions of ordering, is it feasible that a "smallest" object exists? In the case of the values, clearly the subset relation has this property. However, for formulas the situation is more murky. For example, two formulas may entail each other, and therefore are interchangeable in any backup, so backups are not unique. Therefore, we will not be able to compute the "minimal" backup (since this implies a uniqueness we cannot attain.) Instead, we will guarantee one member of the equivalence class of backups under this notion of formula equivalence.[1]

8.4 Computing the Minimal Backup

Using the ordering on backups given above, our goal is to compute the minimal backup. There are two parts of this task: (1) calculating the message content and (2) calculating the relevant formulas. Because the relevant formulas are dependent on the message content—any formulas relevant to identifiers in the message are relevant—we calculate the message content first, then show how to find the relevant formulas.

We will provide a running example by considering the backup at point 8 in the Andrew protocol (Fig. 3.2).

8.4.1 Backup Content

Our general strategy for computing the necessary data to include in the backup of `sl` $\rightarrow$ `sr` is:

[1] Astute observers will recall Church's Theorem, which implies that in reasonable logics "A iff B" is undecidable.

1. Consider the smallest set σ such that $\sigma \vdash \texttt{sr}$.
2. Construct a message, m, that binds σ.
3. Prove that m is well-formed for sending after the execution of `sl`, i.e., condition (1) of a backup.
4. Prove that `sr` is well-formed after receiving m, i.e., condition (2) of a backup.

Computing σ. σ can be computed through a number of proofs and simple manipulations of sets, combined with a lot of tedious work at the bottom-most point. We will walk through the high-level process and leave out the tedium. As a notational convenience, we will write $sset(x)$ for the smallest set σ such that $\sigma \vdash x$, where x may be a strand, strand description, message, or formula. We write $sset_r(x)$ and $sset_s(x)$ for the smallest set such that the message x is well-formed for receiving and sending, respectively.

We first prove the trivial theorem that $sset(.) = \emptyset$ (recall that . is the empty strand). We then prove that for strand description sd and strand s,

$$sset(sd \rightarrow s) = sset(sd) \cup (sset(s) - bound(sd)).$$

The rationale of this theorem is very clear: strand descriptions introduce bindings, which could not possibly be in the backup, and they require bindings, which must be in the backup.

Next, we do a similar case analysis on strand descriptions. For the receiving strand description $-m, fs$, we show that

$$sset(-m, fs) = sset_r(m) \cup (sset(fs) - bound(m)).$$

Intuitively, a message m has data that are necessary to deconstruct it, such as keys or hash contents, ($sset_r(m)$), after which it binds identifiers ($bound(m)$) that are referred to ($sset(fs)$) by the relied-upon formulas, fs. For the sending strand description $+m, fs$, we have a dual formulation where

$$sset(+m, fs) = sset(fs) \cup (sset_s(m) - bound(fs)).$$

Intuitively, a principal guarantees a number of formulas fs, then constructs a message m, which refers to some identifiers ($sset_s(m)$), that may have been bound by the formula derivations ($bound(fs)$).

At the bottom, it remains to define the smallest sets for formulas, sent messages, and received messages. Each of these is very straight-forward and tedious: The identifiers mentioned by a formula are the smallest set for a set of formulas; The identifiers mentioned by a sent message are the smallest set; and, the keys and hash contents are the smallest set for a received message.

Example. In our example, the necessary set is {`chana`, `kabn`, `nbn`, `kab`, `ai`}, because all these are referred to in the strand starting at point 8. In this case, there are no identifiers bound, so nothing is subtracted from the set at any point.

Constructing the message m. It is trivial to construct a message that binds σ: simply include each member of σ in the message, i.e., $m = x_1, x_2, \ldots, x_n$, where x_i is the ith element of σ. We additionally prefix the message with the constant `"backup"` and an integer indicating the point in the protocol. This is a nod to the practical concern that it is necessary to ensure that backups at different points cannot be confused. We formally prove that protocol backups cannot be confused in the sense of Chapter 7.

Example. In the example, the backup message is: `("backup", "8", chana, kabn, nbn, kab, ai)`. In this message, both `nbn` and `ai` are nonces that can be used to distinguish backups of different sessions.

Proving well-formedness of m. We must prove that m is well-formed. If it is not, then an implementation will not be able to generate the backup. Recall that a backup of $\texttt{sl} \rightarrow \texttt{sr}$ is only defined if the strand is well-formed, i.e., $\emptyset \vdash \texttt{sl} \rightarrow \texttt{sr}$. We will use this to prove that the message m is well-formed after `sl`, i.e., it can be generated after `sl`. We first prove the following theorem:

Theorem 17. *(`type_strand_cut`) For strands s and s', $\Sigma \vdash s \rightarrow s'$ if and only if $\Sigma \vdash s$ and $\Sigma \cup bound(s) \vdash s'$.*

Proof summary. Induction on the length of the strand s combined with set theory identities. □

Based on this general result, it is simple to prove that m is well-formed. First, we can read off that $\emptyset \vdash \texttt{sl} \rightarrow \texttt{sr}$ implies $bound(\texttt{sl}) \vdash \texttt{sr}$. Since we know σ is the smallest set such that $\sigma \vdash \texttt{sr}$, we can conclude that $\sigma \subseteq bound(\texttt{sl})$, i.e., the necessary data is present. Then, we prove that $\theta \vdash m$ if and only if $vars(m) \subseteq \theta$, when m is a message that has the form $x_1, x_2, \ldots, x_n$, as our backup message does. Therefore, $bound(\texttt{sl}) \vdash m$, because $vars(m) = \sigma$ and $\sigma \subseteq bound(\texttt{sl})$.

Example. In our example, the message is well-formed because each identifier is bound somewhere before point 8: `chana` is bound at point 1; `kabn` at point 7; `nbn` at point 6; `kab` and `ai` at point 0.

Proving well-formedness of `sr`. Our construction of m allows for a concise proof of $\emptyset \vdash -m, \_ \rightarrow \texttt{sr}$, because we constructed m such that the identifiers bound are exactly $sset(\texttt{sr})$. This is the smallest set σ such that $\sigma \vdash \texttt{sr}$, so clearly $\emptyset \vdash -m, \_ \rightarrow \texttt{sr}$.

Example. In our example, the rest of strand is well-formed because m binds all the identifiers necessary: {`chana`, `kabn`, `nbn`, `kab`, `ai`}.

8.4.2 Computing the formulas

A backup must also entail all formulas relevant to the data. A minimal backup includes that smallest such set. Therefore, we must (a) identify the relevant formulas and (b) compute the minimal set that entails them all.

Relevant formulas. A formula is considered relevant to a value if it contains a reference to its binding. For example, "A is prime" is relevant to the value A. Some formulas mention values not otherwise contained in the backup data. For example, suppose that the backup data is only A, but a formula relevant to A is "$A + B = C$". Since this formula references B and C, they must be included in the backup set as well. Furthermore, any formulas relevant to B and C must also be included.

Thus, the relevant formulas of a backup are an inductively defined set. We must prove that this set is finite and computable. If it is not finite, then it cannot be realistically saved. If it is not computable, then it cannot be constructed, used, or studied.

Theorem 18. *The set of relevant formulas is finite and computable for all backups of all protocols.*

Proof Summary. We rely on the finite length of strands and the finite number of formulas attached to any point in the strand. Since the set of all formulas of a strand is finite and computable, and because identifier reference is computable, we can easily construct the set of relevant formulas. □

Strongest formulas. Once we have the set of relevant formulas, we must select the minimal, or strongest, subset, i.e., the smallest set such that all relevant formulas are entailed.

In our protocol environment, CPPL, the logic of formulas is not specified. Instead, it is left as a parameter, so that whatever logic is necessary for a particular protocol may be used. Therefore, we cannot define the minimality calculation in general, since some logics do not have a computable notion of entailment, and push this burden on the provider of the logic.

However, a very common logic is a simple predicate language combined with a database. This is the logic used by the Andrew Secure RPC example. In this logic, entailment is mapped to inclusion in the database. Therefore, no formula entails anything but itself, and the minimality calculation is trivial: each set is minimal for itself.

8.4.3 Putting Together the Pieces

At this point, we can give our minimum backup of the strand $\mathtt{sl} \rightarrow \mathtt{sr}$: It is the message m that binds $sset(\mathtt{sr})$ as well as all identifiers mentioned in Φ, where Φ is the relevant subset of the formulas that appear in $\mathtt{sl}$.

It is necessary to prove that this backup—m and Φ—satisfy the three conditions on backups: (1) $\emptyset \vdash \mathtt{sl} \rightarrow +m, \Phi \rightarrow .$; (2) $\emptyset \vdash -m, \Phi \rightarrow \mathtt{sr}$; and (3) If x is an identifier mentioned in $\mathtt{sr}$, m, or Φ and f is a formula in $\mathtt{sl}$ that mentions x, then f must be deducible from Φ.

Each of these properties is verified easily by the construction of the various parts. The backup values, m, were constructed such that (1) and (2) hold, and modified in a sound way during the the relevant formula calculation. Property (3) holds by definition of our computation of the relevant formula set.

We present the minimal backup at each point of role B of the Andrew Secure RPC protocol in Table 8.1.

Step	Values	Formulas
0	{}	{}
1	$\{kab, ai\}$	{}
2	$\{kab, ai, chana\}$	$\{accept(chana)\}$
3	$\{kab, ai, chana, na\}$	$\{accept(chana)\}$
4	$\{kab, ai, chana, na, nb\}$	$\{accept(chana), new\_nonce(nb)\}$
5	$\{kab, ai, chana, nb\}$	$\{accept(chana), new\_nonce(nb)\}$
6	$\{kab, ai, chana\}$	$\{accept(chana)\}$
7	$\{kab, ai, chana, nbn\}$	$\{accept(chana), new\_nonce(nbn)\}$
8	$\{kab, ai, chana, nbn, kabn\}$	$\{accept(chana), new\_nonce(nbn), new\_symkey(kabn)\}$
9	$\{ai, kabn\}$	$\{new\_symkey(kabn)\}$

Table 8.1: Andrew Secure RPC Role B Backups

8.5 Insights

As mentioned earlier, the minimal backup of a protocol provides some subtle insights into its structure. This capability comes from a principled understanding of *what* the minimal backup is: an extensional description of what is relevant at each point in the protocol. This extensionality allows us to easily compare and understand protocols.

For example, when the minimal backup at points i and j of a protocol are the same, then the protocol *has the same requirements* at each of those points. This is extremely dangerous, because it means that in principle there is nothing preventing a protocol from skipping directly to point j after point $i-1$. The example protocol, Andrew Secure RPC [122], has this property. Notice in the minimal backup table (Table 8.1) that the backup contents are identical at steps 2 and 6. This means that role B can skip the execution of steps 3 through 5. This directly corresponds to a known attack on the protocol [15].

We learn interesting things about the structure of a protocol even when the minimal backups at two points in the protocol are not identical. Suppose that the only difference between the minimal backups at points 2 and 5 of a protocol is a single identifier, `nb`. This means that the points in between must have been for the very purpose of acquiring `nb`. If this is not actually the case, then something must be wrong with the protocol. An alternative take on this problem is to check when some value is learned *without* the generation of a new nonce to provide an authentication challenge; this pattern reveals the kinds of errors discussed by Lowe [86]. For example, the Denning-Sacco protocol [32] contains only one nonce, rather than two as Lowe suggests. We intend on using this intuition to automatically construct attacks on protocols, but do not pursue that avenue in this work.

This is an example of a general principle: when the size of the minimal backup decreases, then some sub-task has been completed. This reveals the structure of the protocol. In the extreme case, when the size of minimal backup decreases to zero, then this means that the protocol's past is independent of its future. Another way of thinking of this case is that what is presented as *one* protocol is in fact *two* independent

protocols. Naturally, a good cryptographic protocol will not have this property, but many other protocols do: for example, after a request-response sequence in HTTP, the protocol essentially starts afresh. Our analysis would show this as the minimal backup becoming empty.

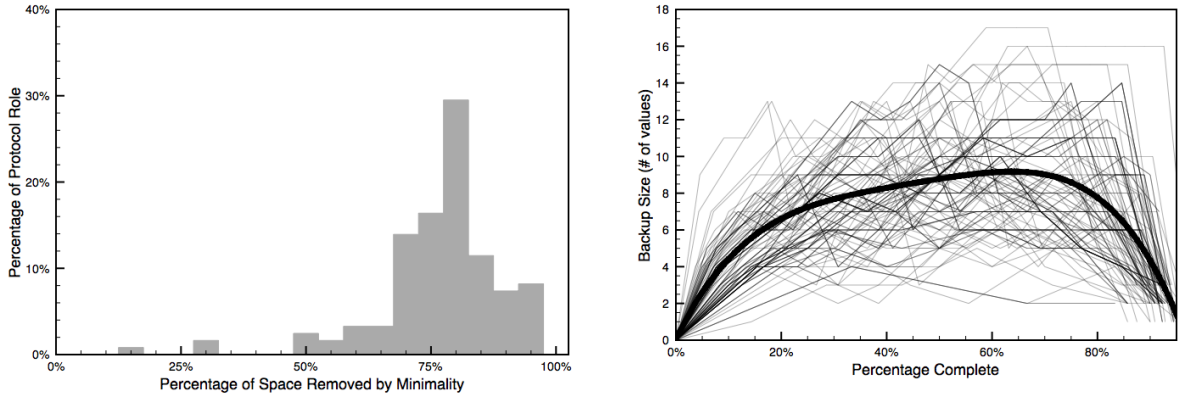
8.6 Applications

Although we present minimal backups primarily as a diagnostic tool, they are useful in practical applications. The most obvious application of minimal backups is, of course, as a backup. If the minimal backup is stored on secure, non-volatile storage, then it can be used to recover a running protocol session from a failure. Job migration is a natural out-cropping of this application, as migration can be understood as causing a "failure" and "recovering" from it at another location. Of course, in a deployment where the protocol is part of some larger service, it is also necessary to handle TCP state (e.g., [92, 127]), persistent service data [80], et cetera [36], but the minimal backup of the cryptographic protocol provides the foundation of the entire fault-tolerance milieu. Both of these applications provide core components of a scalability strategy designed to handle high customer load: backups deal with inevitable failure and job migration enables load-balancing.

8.7 Evaluation

We have studied the minimal backups of 121 protocol roles found in the SPORE Protocol Repository [114]. We have used our analysis to determine the minimal backups at every point in each of these protocols. Our implementation, extracted from Coq, was able to calculate these in approximately 1 second. We were interested in the applicability of the minimal backup-based analysis and the pragmatic benefits of minimality.

First, we found that no protocol role had an empty minimal backup after the first step of the protocol. Second, we found that every protocol exhibited some decrease in the size of the minimal backup at some point in the protocol. We calculated the largest percentage decrease in backup size for each protocol and determined the minimum (16%), mean (65%), median (69%), and maximum (93%). This shows that most protocols discard over half of their knowledge at some point in the run. Figure 8.2a shows the space savings due to minimality as a percentage of total backup space. Notice that the graph is heavily weighted to the right: this demonstrates the practical utility of minimality. Figure 8.2b presents a graph of the size of a minimal backup of a protocol at each stage of completion, where the average of all protocols has been emphasized. This graph indicates that backups have an "n" shape, which matches the intuition that they start from a small amount of knowledge, build towards some goal, but no longer require the intermediate knowledge by the end of the run. This evidence of our intuition justifies the diagnostic application of the minimal backup graph.



(a) (b)

Figure 8.2: Minimal Backup Diagnostic Graphs

8.8 Related Work

Analysis. Our analysis is similar to live expression analysis [73]. Liveness analysis determines whether a value is necessary in the rest of a computation, similar to how our analysis determines what information is necessary to complete a protocol run. We could use liveness analysis on the implementation of a cryptographic protocol to get some of the results of our analysis, e.g., sound backups. However, liveness analysis is typically not complete for implementation languages. Our analysis is complete, modulo our approximation of entailment[2], in the form of our proof of minimality of backups.

If we used live-expression analysis at the implementation level, then we would approximate the checkpointing approach of Li et al.'s [79]. This approach instruments a program with points where full system checkpoints can be taken. Due to the difficulty in determining what data is live in c, they must snapshot the entire system memory. They pursue minimality via a heuristic training process that determines where to place checkpoint locations. While their work is inspiring, we do not need to resort to these techniques, because we can calculate provably minimal backups. It would be interesting to compare the empirical results of each approach to ascertain how close they come to minimality.

Diagnostic. The diagnostic aspect of minimal backups can be seen as a generalization of the idea of authentication tests [51]. These are protocol patterns that represent the attainment of a sub-goal. In a sense, analyzing the minimal backup is a means of finding other similar patterns. However, minimal backup patterns do not provide the guarantees of an authentication test, in general.

The diagnostic aspect of our analysis can also be considered a way of determining the degree to which a protocol exhibits soft-state [115, 67, 80]. This state, recognized as desirable by systems designers, is what is not *necessary* for a protocol to execute properly, but may be useful in improving performance. For example, by definition a cache protocol does not need a cached version of all of memory, so it can remove pages from

[2]In our implementation, the logic's entailment relation *is* decidable. This caveat only applies when an undecidable logic is used.

the cache under press. But when it has some particular page, there are performance benefits. In a fuzzy way, the difference between the entire state and the minimal backup represents the soft-state. Our literature review has not unveiled any analyses that automatically determine the degree to which a protocol uses soft-state.

Fault-tolerance. In the realm of cryptographic protocols, Williams [142] discusses how to produce a fault-tolerant version of a trusted third-party authentication service, with applications to Kerberos and the Needham-Schroeder protocol. At the time of this writing, we were not able to obtain a copy of this paper, despite email requests. Based on the abstract, we assume that (a) their work is tied to details about the protocols in question, rather than a general technique; and, (b) that they do not provide verified guarantees about backup minimality.

Gong [43] developed a distribution authentication protocol designed to increase fault-tolerance, while providing reasonable security guarantees. Inspired by this approach, Reiter [116, 117] developed a methodology of constructing replications of general services and applied this work to an authentication protocol.

These two strategies work by requiring multiple servers to partially serve, or authenticate, a client that has many such servers available. The strategy increases fault-tolerance, because only a fraction of those servers must be available at any given time. Our work contrasts with this approach by providing an analysis that allows fail-over of *any* protocol without modification through a uniform strategy of storing the minimal backup securely.

8.9 Future Work

We are interested in investigating how to automatically analyze minimal backups to identify common, or important, protocol structures. For example, we might be able to automatically identify authentication tests or replay attacks. We are also interested in pursuing an optimization of cryptographic protocols that decreases the size of the minimal backup, while preserving the security guarantees of each protocol role.

8.10 Conclusion

We have formally defined a protocol backup for a particular point in a cryptographic protocol execution. We have shown how to provably compute the minimal backup, given a reasonable ordering on backups. We have described the utility of the minimal backup as a diagnostic to protocol designers. We have also explained how the minimal backup can be used to solve the problems of fault-tolerance and job-migration of cryptographic protocol sessions. Finally, we have provided a summary of our study of the minimal backups of 121 protocol roles from the SPORE repository.

This work demonstrates that the internal structure of a protocol is revealed by investigating its minimal backup. This is made possible by the formal proof that the backups are truly minimal and that their construction is sound with respect to the protocol semantics. This analysis is also useful for enabling a scalability

property: fault-tolerance.

Note. A paper based on the content of this chapter appeared at Formal Methods in Security Engineering workshop in 2008.

Part IV

Epilogue

Chapter 9

Related Work

As we have discussed each portion of our work, we have related it to the literature:

1. WPPL: Section 6.4
2. Dispatching: Section 7.7
3. Minimal Backups: Section 8.8

There is some work similar to the broad stroke of our work, however. In particular, there are a number of attempts at creating domain-specific language for network protocol implementations. For example, Mace [74], Flux [14], the Austin Protocol Compiler [95], Prolac [76] and other systems [56, 82, 5, 125, 59, 62, 68, 138, 31] each define a language for expressing protocol definitions and compiling correct, secure, and efficient implementations. However, in each of these projects, the goal is to (a) avoid common bugs related to buffer overflows and concurrency; (b) abstract away details of queuing and concurrency, so different methodologies can be applied; and (c) ensure greater implementation compliance with a protocol specification. In addition, the protocols under discussion are ad-hoc IETF-style protocols without formal specifications that have a very different set of assumptions. In contrast, our goal is to analyze protocols and render more information that can be used, in a sense, for smarter optimizations of such implementations. For these reasons, our work is supplementary to these efforts: we can easily leverage them to improve our implementations.

Additionally, a number of systems [91, 89, 112, 93] exist to provide high-level specifications of packet formats and compile efficient packet parsers and emitters. We could use the strategies of these systems in our work at a low-level, or apply them as black-boxes, but we cannot rely on them for the higher-level issues such as fault-tolerance and dispatching.

Chapter 10

Future Work

As we have discussed each portion of our work, we have discussed its future opportunities:

1. WPPL: Section 6.5
2. Dispatching: Section 7.8
3. Minimal Backups: Section 8.9

There are, however, future directions this work could take that are at a higher level. We would like to investigate when it is safe to modify cryptographic protocols. A major assumption of our work has been that it is not advisable to do this, but it is clearly possible in many cases. This would allow us to understand what effect extant transformations have on a protocol's security and scalability properties. Furthermore, it would allow us to develop transformation that preserve security proofs, while improving the scope of analysis or deployment.

Chapter 11

Conclusion

Static analyses of cryptographic protocols can provide insight into the non-security properties of a protocol and aid in the deployment and compilation of implementations. Protocol designers use a specification style that is incompatible with most existing protocol programming languages; a domain-specific language that matches this style will improve the usability of protocol analyses.

We have shown two different families of static analyses that provide insight into the properties of protocols.

First, our message space analysis (Ch. 7) determines when a pair of protocols' message spaces overlap. This can be applied to determine when a protocol role can support many concurrent users or when a pair of protocol roles can be deployed concurrently. We have also shown that the trust optimization that determines how many secrets are necessary to guarantee distinct message spaces reveals why two spaces are distinct. This optimization can also be used to efficiently and correctly perform dispatching in an implementation.

Second, our minimal backup analysis (Ch. 8) determines the necessary information at each point in a protocol role's execution. This information can be used to study the structure of the protocol and determine how it satisfies its goals. This information can also serve an optimal session backup. These backups can be used to provide a fault-tolerant implementation with check-pointing. Backups are also useful for performing job migration in a load-balancing system.

The benefits of both of these analyses come without cost to the trustworthiness or soundness of the protocol implementation. The formal, mechanical proofs about the soundness of these analyses guarantees (1) conclusions based on the analysis are valid with respect to the protocol and (2) that the protocol role has the same meaning under dispatching or after backups, etc.

These analyses operate on CPPL (Ch. 4), a domain-specific language, that does not match the specification style of protocol designers. WPPL (Ch. 6) is a domain-specific language that *does* match this style (evidenced by being able to encoding the SPORE repository.) WPPL specifications can be projected into CPPL end-points in a meaning-preserving way. This means that all analyses of CPPL protocols can be performed on WPPL protocols.

This greater usability of the analyses does not come at a cost due to formally verified end-point projection.

There we conclude that static analyses of cryptographic protocols can provide insight into the non-security properties of a protocol and aid in the deployment and compilation of implementations. A domain-specific language that matches the style of protocol designers improves the usability of protocol analyses.

Bibliography

[1] IEEE 802.11 Local and Metropolitan Area Networks: Wireless LAN Medium Acess Control (MAC) and Physical (PHY) Specifications, 1999.

[2] Martin Abadi. Security protocols and their properties. In *Foundations of Secure Computation*, 2000.

[3] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, January 1999.

[4] Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, January 1996.

[5] Mark B. Abbott and Larry L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Trans. Netw.*, 1(1):4–19, 1993.

[6] Alessandro Armando, David A. Basin, Mehdi Bouallagui, Yannick Chevalier, Luca Compagna, Sebastian Mödersheim, Michaël Rusinowitch, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISS security protocol analysis tool. In *Computer Aided Verification*, 2002.

[7] Andrew Begel, Steven McCanne, and Susan L. Graham. BPF+: exploiting global data-flow optimization in a generalized packet filter architecture. In *Symposium on Communications, Architectures and Protocols*, 1999.

[8] Chiara Bodei, Mikael Buchholtz, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.

[9] Chiara Bodei, Mikael Buchholtz, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Automatic validation of protocol narration. In *Computer Security Foundations Workshop*, 2003.

[10] Chiara Bodei, Pierpaolo Degano, Han Gao, and Linda Brodo. Detecting and preventing type flaws: a control flow analysis with tags. *Electronic Notes in Theoretical Computer Science*, 194(1):3–22, 2007.

[11] C. Boyd. Hidden assumptions in cryptographic protocols. *Computers and Digital Techniques, IEE Proceedings* -, 137(6):433–436, Nov 1990.

[12] Sébastien Briais and Uwe Nestmann. A formal semantics for protocol narrations. *Theoretical Computer Science*, 389(3):484–511, 2007.

[13] J. Bull and D. J. Otway. The authentication protocol. Technical Report DRA/CIS3/PROJ/CORBA/SC/1/CSM/436-04/03, Defence Research Agency, 1997.

[14] Brendan Burns, Kevin Grimaldi, Alexander Kostadinov, Emery D. Berger, and Mark D. Corner. Flux: A language for programming high-performance servers. In *USENIX*, pages 129–142, 2006.

[15] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *Proceedings of the Royal Society*, Series A, 426(1871):233–271, December 1989.

[16] C. Caleiro, L. Viganò, and D. Basin. Deconstructing Alice and Bob. *Electronic Notes in Theoretical Computer Science*, 135(1):3–22, 2005.

[17] C. Caleiro, L. Viganò, and D. Basin. On the semantics of Alice&Bob specifications of security protocols. *Theoretical Computer Science*, 367(1-2):88–122, 2006.

[18] Carlos Caleiro, Luca Vigano, and David Basin. Relating Strand Spaces and Distributed Temporal Logic for Security Protocol Analysis. *Logic Jnl IGPL*, 13(6):637–663, 2005.

[19] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In *European Symposium on Programming*, 2007.

[20] CCITT. The directory authentification framework. Draft Recommendation X.509, 1987. Version 7.

[21] I. Cervesato, N. Durgin, M. Kanovich, and A. Scedrov. Interpreting strands in linear logic. In *Workshop on Formal Methods and Computer Security*, 2000.

[22] I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Relating strands and multiset rewriting for security protocol analysis. In *Proceedings, 13th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, July 2000.

[23] John Clark and Jeremy Jacob. On the security of recent protocols. *Information Processing Letters*, 56(3):151–155, November 1995.

[24] John Clark and Jeremy Jacob. A survey of authentication protocol literature, November 1997.

[25] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*, 2006.

[26] Ricardo Corin, Pierre-Malo Denielou, Cedric Fournet, Karthikeyan Bhargavan, and James Leifer. Secure implementations for typed session abstractions. In *Computer Security Foundations Symposium*, 2007.

[27] Véronique Cortier, Jérémie Delaitre, and Stéphanie Delaune. Safely Composing Security Protocols. In *Conference on Foundations of Software Technology and Theoretical Computer Science*, 2007.

[28] R. Cramer, I. Damgård, and B. Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *Proc. 14th Annual International Cryptology Conference (CRYPTO'94)*, volume 963 of *LNCS*, pages 174–187, Santa Barbara (California, USA), 1994. Springer-Verlag.

[29] Federico Crazzolara and Glynn Winskel. Petri nets in cryptographic protocols. In *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*, page 149, Washington, DC, USA, 2001. IEEE Computer Society.

[30] C.J.F. Cremers. Feasibility of multi-protocol attacks. In *International Conference on Availability, Reliability and Security*, 2006.

[31] Walid Dabbous, Sean O'Malley, and Claude Castelluccia. Generating efficient protocol code from an abstract specification. In *SIGCOMM*, pages 60–72, New York, NY, USA, 1996. ACM.

[32] Dorothy Denning and G. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8), August 1981.

[33] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[34] Shaddin F. Doghmi, Joshua D. Guttman, and F. Javier Thayer. Skeletons, homomorphisms, and shapes: Characterizing protocol executions. *Electronic Notes in Theoretical Computer Science*, 173:85–102, 2007.

[35] Daniel Dolev and Andrew Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 29:198–208, 1983.

[36] Fred Douglis. Transparent process migration: Design alternatives and the Sprite implementation. *Software–Practice and Experience*, 21(8):757–785, August 1991.

[37] Antonio Durante, Riccardo Focardi, and Roberto Gorrieri. A compiler for analyzing cryptographic protocols using noninterference. *ACM Transactions on Software Engineering and Methodology*, 9(4):488–528, 2000.

[38] Michael D. Ernst, Craig S. Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *European Conference on Object-Oriented Programming*, pages 186–211, 1998.

[39] F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Why is a security protocol correct? In *IEEE Symposium on Security and Privacy*, 1998.

[40] Alan Frier, Philip Karlton, and Paul Kocher. The SSL 3.0 protocol. Internet Draft, November 1996.

[41] J. A. Garay, M. Jakobsson, and P. MacKenzie. Abuse-free optimistic contract signing. In *International Cryptology Conference*, 1999.

[42] Li Gong. Using one-way functions for authentication. *Computer Communication Review*, 19(5):8–11, October 1989.

[43] Li Gong. Increasing availability and security of an authentication service. *IEEE Journal on Selected Areas in Communications*, 11(5):657–662, 1993.

[44] Joshua D. Guttman. Key compromise and the authentication tests. *Electronic Notes in Theoretical Computer Science*, 47, 2001.

[45] Joshua D. Guttman. Security goals: Packet trajectories and strand spaces. In Roberto Gorrieri and Riccardo Focardi, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*, pages 197–261. Springer Verlag, 2001.

[46] Joshua D. Guttman. Security protocol design via authentication tests. In *Proceedings, 15th Computer Security Foundations Workshop*. IEEE Computer Society Press, June 2002.

[47] Joshua D. Guttman. Authentication tests and disjoint encryption: a design method for security protocols. *Journal of Computer Security*, 12(3/4):409–433, 2004.

[48] Joshua D. Guttman, Jonathan C. Herzog, John D. Ramsdell, and Brian T. Sniffen. Programming cryptographic protocols. In *Trust in Global Computing*, 2005.

[49] Joshua D. Guttman and F. Javier Thayer. Protocol independence through disjoint encryption. In *Computer Security Foundations Workshop*, 2000.

[50] Joshua D. Guttman and F. Javier Thayer. Authentication tests and the structure of bundles. *Theoretical Computer Science*, 283(2):333–380, June 2002.

[51] Joshua D. Guttman and F. Javier Thayer. Authentication tests and the structure of bundles. *Theoretical Computer Science*, 283(2):333–380, June 2002.

[52] Joshua D. Guttman and F. Javier Thayer. The sizes of skeletons: Decidable cryptographic protocol authentication and secrecy goals. MTR 05B09 Revision 1, The MITRE Corporation, March 2005.

[53] Joshua D. Guttman, F. Javier Thayer, Jay A. Carlson, Jonathan C. Herzog, John D. Ramsdell, and Brian T. Sniffen. Trust management in strand spaces: A rely-guarantee method. In *European Symposium on Programming*, 2004.

[54] Christian Haack and Alan Jeffrey. Pattern-Matching Spi-Calculus. In *Formal Aspects in Security and Trust*, pages 55–70, 2004.

[55] Joseph Y. Halpern and Riccardo Pucella. On the relationship between strand spaces and multi-agent systems. *ACM Transactions on Information and System Security*, 6(1):43–70, February 2003.

[56] Kenta Hatori and Kei Hiraki. A network programming language based on concurrent processes and regular expressions. In *SE'07: Proceedings of the 25th conference on IASTED International Multi-Conference*, pages 105–110, Anaheim, CA, USA, 2007. ACTA Press.

[57] James Heather. Strand spaces and rank functions: More than distant cousins. In *CSFW '02: Proceedings of the 15th IEEE workshop on Computer Security Foundations*, page 104, Washington, DC, USA, 2002. IEEE Computer Society.

[58] James Heather, Gavin Lowe, and Steve Schneider. How to prevent type flaw attacks on security protocols. In *Computer Security Foundations Workshop*, 2000.

[59] Diane Hernek and David P. Anderson. Efficient automated protocol implementation using rtag. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1989.

[60] Jonathan Herzog, Joshua Guttman, and Fred Chase. A strand space analysis of the SSH version 2 protocol. MITRE Product MP 98B0000056, September 1998.

[61] Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 2000.

[62] Bernd Hofmann and Wolfgang Effelsberg. Efficient implementation of estelle specifications. Technical report, University of Mannheim, 1993.

[63] William A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980. Reprint of 1969 article.

[64] Tzonelih Hwang and Yung-Hsiang Chen. On the security of splice/as : The authentication system in wide internet. *Information Processing Letters*, 53:97–101, 1995.

[65] Tzonelih Hwang, Narn-Yoh Lee, Chuang-Ming Li, Ming-Yung Ko, and Yung-Hsiang Chen. Two attacks on Neuman-Stubblebine authentication protocols. *Information Processing Letters*, 53:103–107, 1995.

[66] Florent Jacquemard, Michaël Rusinowitch, and Laurent Vigneron. Compiling and verifying security protocols. *Logic for Programming and Automated Reasoning*, 2000.

[67] Ping Ji, Zihui Ge, Jim Kurose, and Don Towsley. A comparison of hard-state and soft-state signaling protocols. In *SIGCOMM Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2003.

[68] nas José A. Ma and Tomás de Miguel. From lotos to c. In *Proceedings of the First International Conference on Formal Description Techniques*, pages 79–84, Amsterdam, The Netherlands, The Netherlands, 1989. North-Holland Publishing Co.

[69] I Lung Kao and Randy Chow. An efficient and secure authentication protocol using uncertified keys. *Operating Systems Review*, 29(3):14–21, 1995.

[70] Jonathan Katz and Moti Yung. Scalable protocols for authenticated group key exchange. In *CRYPTO*, pages 110–125, 2003.

[71] Axel Kehne, Jürgen Schönwälder, and Horst Langendörfer. Multiple authentications with a nonce-based protocol using generalized timestamps. In *Proc. ICCC '92*, Genua, 1992.

[72] Andrew J. Kennedy. Functional pearl: Pickler combinators. *Journal of Functional Programming*, 14(6):727–739, 2004.

[73] Gary A. Kildall. A unified approach to global program optimization. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1973.

[74] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: language support for building distributed systems. *SIGPLAN Not.*, 42(6):179–188, 2007.

[75] J. Kohl and C. Neuman. The Kerberos network authentication service (v5). RFC 1510, September 1993.

[76] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. A readable tcp in the prolac protocol language. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 3–13, New York, NY, USA, 1999. ACM.

[77] Eddie Kohler, Robert Morris, and Benjie Chen. Programming language optimizations for modular router configurations. In *Architectural Support for Programming Languages and Operating Systems*, 2002.

[78] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *Programming Language Design and Implementation*, 1996.

[79] Chung-Chi Jim Li, Elliot M. Stewart, and W. Kent Fuchs. Compiler-assisted full checkpointing. *Software–Practice and Experience*, 24(10):871–886, 1994.

[80] Benjamin C. Ling, Emre Kiciman, and Armando Fox. Session state: Beyond soft state. In *Networked Systems Design and Implementation*, 2004.

[81] Benjamin W. Long, Colin J. Fidge, and David A. Carrington. Cross-layer verification of type flaw attacks on security protocols. In *Australasian Computer Science Conference*, 2007.

[82] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. *SIGOPS Oper. Syst. Rev.*, 39(5):75–90, 2005.

[83] Gavin Lowe. An attack on the Needham-Schroeder public key authentication protocol. *Information Processing Letters*, 56(3):131–136, November 1995.

[84] Gavin Lowe. Some new attacks upon security protocols. In *Computer Security Foundations Workshop*, 1996.

[85] Gavin Lowe. Casper: A compiler for the analysis of security protocols. In *Computer Security Foundations Workshop*, 1997.

[86] Gavin Lowe. A family of attacks upon authentication protocols. Department of Mathematics and Computer Science 5, University of Leicester, 1997.

[87] Gavin Lowe. A hierarchy of authentication specifications. In *10th Computer Security Foundations Workshop Proceedings*, pages 31–43. IEEE Computer Society Press, 1997.

[88] Gavin Lowe. Toward a completeness result for model checking of security protocols. In *Computer Security Foundations Workshop*, pages 96–105. IEEE Computer Society Press, 1998.

[89] Anil Madhavapeddy, Alex Ho, Tim Deegan, David Scott, and Ripduman Sohan. Melange: creating a "functional" internet. *SIGOPS Oper. Syst. Rev.*, 41(3):101–114, 2007.

[90] Dalia Malki and Michael K. Reiter. A high-throughput secure reliable multicast protocol. In *CSFW*, pages 9–17, 1996.

[91] Yitzhak Mandelbaum, Kathleen Fisher, David Walker, Mary Fernandez, and Artem Gleyzer. Pads/ml: a functional data description language. *POPL*, 42(1):77–83, 2007.

[92] M. Marwah, S. Mishra, and C. Fetzer. Tcp server fault tolerance using connection migration to a backup server. In *Dependable Systems and Networks*, 2003.

[93] Peter J. McCann and Satish Chandra. Packet types: abstract specification of network protocol messages. In *Symposium on Communications, Architectures and Protocols*, 2000.

[94] Jay McCarthy, Joshua D. Guttman, John D. Ramsdell, and Shriram Krishnamurthi. Compiling cryptographic protocols for deployment on the Web. In *World Wide Web*, pages 687–696, 2007.

[95] Tommy Marcus Mcguire. *Correct implementation of network protocols*. PhD thesis, University of Texas at Austin, 2004.

[96] Catherine Meadows. A model of computation for the NRL protocol analyzer. In *Computer Security Foundations Workshop*, 1994.

[97] Catherine Meadows. Analyzing the Needham-Schroeder public-key protocol: A comparison of two approaches. In *European Symposium on Research in Computer Security*, 1996.

[98] Catherine Meadows. Identifying potential type confusion in authenticated messages. In *Computer Security Foundations Workshop*, 2002.

[99] Jonathan Millen and Frederic Muller. Cryptographic protocol generation from CAPSL. Technical Report SRI-CSL-01-07, SRI International, December 2001.

[100] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, 1979.

[101] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst.*, 15(3):217–252, 1997.

[102] Judy H. Moore. Protocol failures in cryptosystems. *Proceedings of the IEEE*, 76(5), May 1988.

[103] Tom Murphy VII, Karl Crary, and Robert Harper. Type-safe distributed programming with ML5. In *Trustworthy Global Computing 2007 pre-proceedings*, November 2007.

[104] Roger Needham and Michael Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12), 1978.

[105] Roger M. Needham and M. D. Schroeder. Authentication revisited. *Operating Systems Review*, 21(7):7–7, January 1987.

[106] Matthias Neubauer and Peter Thiemann. From sequential programs to multi-tier applications by program transformation. In *Principles of Programming Languages*, pages 221–232, 2005.

[107] B. Clifford Neuman and Theodore Ts'o. Kerberos : An authentication service for computer networks. Technical Report ISI/RS-94-399, USC/ISI, 1994.

[108] B. Clifford Neumann and Stuart G. Stubblebine. A note on the use of timestamps as nonces. *Operating Systems Review*, 27(2):10–14, april 1993.

[109] Greg O'Shea and Michael Roe. Child-proof authentication for MIPv6 (CAM). *Computer Communications Review*, April 2001.

[110] D. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, January 1987.

[111] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Io-lite: a unified i/o buffering and caching system. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 15–28, Berkeley, CA, USA, 1999. USENIX Association.

[112] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. binpac: a yacc for writing application protocol parsers. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 289–300, New York, NY, USA, 2006. ACM.

[113] Lawrence C. Paulson. Relations between secrets: Two formal analyses of the Yahalom protocol. *Journal of Computer Security*, 2001.

[114] Project EVA. Security protocols open repository. http://www.lsv.ens-cachan.fr/spore/, 2007.

[115] Suchitra Raman and Steven McCanne. A model, analysis, and protocol framework for soft state-based communication. In *SIGCOMM Applications, Technologies, Architectures, and Protocols for Computer Communications*, 1999.

[116] Michael K. Reiter and Kenneth P. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems*, 16(3):986–1009, 1994.

[117] Michael K. Reiter, Kenneth P. Birman, and Robbert van Renesse. A security architecture for fault-tolerant systems. *ACM Transactions on Computer Systems*, 12(4):340–371, 1994.

[118] J. Robin, B. Cockett, and J. A. Hierrera. Decision tree reduction. *Journal of ACM*, 37(4):815–842, 1990.

[119] A. W. Roscoe. Modeling and verifying key-exchange protocols using CSP and FDR. In *Computer Security Foundations Workshop*, pages 98–107, 1995.

[120] P. Y. A. Ryan and S. A. Schneider. An attack on a recursive authentication protocol: A cautionary tale. *Information Processing Letters*, 65(1):7–10, 1998.

[121] Khair Eddin Sabri and Ridha Khedri. A mathematical framework to capture agent explicit knowledge in cryptographic protocols. Technical Report CAS-07-04-RK, McMaster University, 2007.

[122] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7(3):247–280, 1989.

[123] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop, a language for programming the web 2.0. In *Scheme and Functional Programming*, 2006.

[124] Victor Shoup and Avi Rubin. Session key distribution using smart cards. In *In Proceedings of Advances in Cryptology, EUROCRYPT'96*, volume 1070 of *LNCS*. Springer-Verlag, 1996.

[125] Deepinder Sidhu and Anthony Chung. A formal description technique for protocol engineering. Technical report, University of Maryland at College Park, 1990.

[126] Dawn Xiaodong Song. Athena: a new efficient automated checker for security protocol analysis. In *Computer Security Foundations Workshop*, 1999.

[127] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory tcp: connection migration for service continuity in the internet. *Distributed Computing Systems*, 2002.

[128] Paul Syverson. Towards a strand semantics for authentication logic. *Electronic Notes in Theoretical Computer Science*, 20, 1999.

[129] Paul Syverson and Catherine Meadows. A logic language for specifying cryptographic protocol requirements. In *Research in Security and Privacy*, pages 165–177, May 1993.

[130] M. Tatebayashi, N. Matsuzaki, and D.B. Newman. Key distribution protocol for digital mobile communication systems. In *Advance in Cryptology — CRYPTO '89*, volume 435 of *LNCS*, pages 324–333. Springer-Verlag, 1989.

[131] F. Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2/3):191–230, 1999.

[132] F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Honest ideals on strand spaces. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, June 1998.

[133] F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Mixed strand spaces. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, June 1999.

[134] The Coq development team. *The Coq proof assistant reference manual*, 8.1 edition, 2007.

[135] Thomson. Smartright technical white paper v1.0. Technical report, Thomson, october 2001.

[136] Laurent Vigneron. Positive deduction modulo regular theories. In *Computer Science Logic*, pages 468–485, 1996.

[137] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *HotOS*, 2003.

[138] S. T. Vuong, A. C. Lau, and R. I. Chan. Semiautomatic implementation of protocols using an estelle-c compiler. *IEEE Trans. Softw. Eng.*, 14(3):384–393, 1988.

[139] David Wagner. Cryptographic protocols for electronic voting. In *CRYPTO*, page 393, 2006.

[140] Mark Weiser. Program slicing. In *International Conference on Software Engineering*, 1981.

[141] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001.

[142] D. Williams and H. Lutfiyya. Fault-tolerant authentication services. *International Journal of Computers and Applications*, 2007.

[143] Thomas Y. C. Woo and Simon S. Lam. A lesson on authentication protocol design. *Operating Systems Review*, pages 24–37, 1994.

[144] Suguru Yamaguchi, Kiyohiko Okayama, and Hideo Miyahara. The design and implementation of an authentication system for the wide area distributed environment. *IEICE Transactions on Information and Systems*, E74(11):3902–3909, November 1991.

[145] Fan Yang, Nitin Gupta, Nicholas Gerner, Xin Qi, Alan Demers, Johannes Gehrke, and Jayavel Shanmugasundaram. A unified platform for data driven web applications with automatic client-server partitioning. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 341–350, New York, NY, USA, 2007. ACM.

[146] T. Ylonen, T. Kivinen, and M. Saarinen. SSH authentication prototcol. Internet draft, November 1997. Also named draft-ietf-secsh-userauth-01.txt.